Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications

Kumseok Jung*[1] | Julien Gascon-Samson[2] | Shivanshu Goyal[1] | Armin Rezaiean-Asel[1] | Karthik Pattabiraman[1]

[1]Electrical and Computer Engineering Department, University of British Columbia, Vancouver, Canada
[2]Software and IT Engineering Department, École de technologie supérieure, Montreal, Canada

Correspondence
*Kumseok Jung, Corresponding address. Email: kumseok@ece.ubc.ca

Present Address
Present address

Abstract

The Internet of Things (IoT) has gained wide popularity both in academic and industrial contexts. Unlike traditional embedded devices with specialized firmwares, modern IoT devices accommodate general-purpose operating systems, allowing developers to run more sophisticated applications written in high-level languages like JavaScript.

Because IoT devices are subject to resource constraints like available battery power, we need to dynamically migrate a running process between different devices to prevent losing state. However, it is challenging to apply migration techniques using memory snapshots across the heterogeneous pool of IoT devices.

We present ThingsMigrate, a middleware providing platform-independent migration of JavaScript processes across IoT devices. Prior to execution, ThingsMigrate instruments the source code of a given program to expose its internal state. During run-time, the transformed program produces on demand a JSON snapshot of its current state, from which new code is generated to resume execution. Thus, ThingsMigrate enables process migration entirely in the application space without any modifications to the underlying Virtual Machine (VM), providing VM-independence.

We present 3 versions of ThingsMigrate, each building on the previous to optimize for run-time latency and memory consumption. We report on the experience of building each successive version and discuss the insights gained and the learning outcomes.

We evaluated ThingsMigrate against standard benchmarks, over two IoT platforms and a cloud-like environment. We show that it can migrate even highly CPU-intensive applications, with average run-time latency overhead of 33% and memory overhead of 78%. ThingsMigrate supports multiple subsequent migrations without introducing additional overhead over each subsequent migration.

KEYWORDS:
JavaScript, Process Migration, Program Transformation, Internet of Things

1 | INTRODUCTION

The Internet of Things (IoT) comprises a diverse range of devices across different domains processing and exchanging data over the Internet. Over the last few years, the IoT market has grown considerably, with some estimates putting the number of IoT devices to grow to tens of billions[1,2]. One of the factors contributing to this growth is the increased programmability of devices[2,3], as more devices – such as the popular Raspberry Pi – are equipped with general-purpose operating systems in contrast to the traditional embedded devices with specialized firmwares. As a result, developers other than the device manufacturers can write custom software to run on arbitrary IoT devices and device users can install third-party software, giving rise to a flourishing software market much like the smartphone app market.

1.1 | Motivation

JavaScript is prevalent in IoT. In such a software ecosystem, high-level languages running on a Virtual Machine (VM) – e.g., JavaScript, Python – offer many advantages such as greater code portability/reusability and developer productivity, due to the platform-independent semantics and higher level of abstraction than low-level languages like C or Assembly. In this paper, we focus on JavaScript as the programming language for IoT applications.

While JavaScript has enjoyed wide popularity in the Web for a long time, it is now a mature and rich general-purpose programming language. JavaScript is one of the most popular languages today (in 2020), and ranks seventh in the TIOBE programming languages index[4]. It has also been ranked as the top language on popular open-source development communities such as GitHub and Stack Overflow for the last eight years.

More recently, it has become more prevalent in the IoT domain[5,6] following the widespread adoption of Node.js as a server-side language. In fact, the use of JavaScript opens the possibility of sharing a common codebase and data formats (e.g., JSON) across the Web and the IoT software stacks (e.g., the client-side and server-side portions of end-user applications in a Web of Things (WoT)[7] setting could both be written in JavaScript[8,9,10]). As many IoT devices nowadays provide a browser-based interface, it is fair to assume that they integrate a JavaScript VM. In fact, there are efforts being made at either adopting existing JavaScript VMs (e.g., Node.js for IoT devices[11]), or developing new JavaScript VMs[5,6,12,13] for the IoT.

What makes JavaScript particularly attractive in the context of IoT is its single-threaded, asynchronous, and event-driven execution model. Programming concurrent computations (e.g., Actor model) in JavaScript is fairly straightforward using continuation-passing style (CPS) patterns, where a closure is passed to a function invocation as a callback. The closure carries the execution context for each chain of function calls, and the programmer does not need to worry about implementing mutex or semaphores. Furthermore, the asynchronous execution model means that concurrent function calls are interleaved in the JavaScript event queue, eliminating any sleep time and maximizing CPU utilization. The event-driven paradigm maps well to the IoT landscape, as sensors can be expressed as event emitters and actuators as event listeners.

To sum up, the programming model and the broad applicability of JavaScript makes it an appealing target language for IoT[14]. Popular IoT frameworks such as Samsung SmartThings, Node-RED[15], and AWS Greengrass[16] have all chosen JavaScript as one of the main programming languages for userland applications.

IoT applications are becoming stateful. With the growing popularity and programmability of IoT devices, the idea of *edge/-fog computing* is making a comeback[17,18], where the compute workload is placed near the IoT nodes or on the IoT devices themselves. The *edge computing* paradigm offers certain advantages in terms of latency, bandwidth consumption, and some aspects of security[18]. We envision running on the IoT devices more complex applications that were traditionally run on the *cloud*, which would render better utilization of the resources in the underlying computer network. Consequently, as IoT devices and applications become more complex, they inherently generate more elements of *state* (i.e., variables, arrays, objects) during run-time. For instance, in an application that detects motion patterns in a video stream (Section 7), elements of state would include the pixels of the video frames being inspected, as well as any intermediate results produced as part of the computation.

IoT devices are resource-constrained. While general purpose OSes and high-level language VMs may bring programmability to IoT, IoT systems are under different constraints than traditional PC networks or *cloud* datacenters. For instance, many IoT devices are battery powered – such as wearables and drones – meaning that they have a finite timespan for doing useful work and their compute capacity is influenced by the energy available[19]. Compute resources such as memory and CPU cores are more scarce than *cloud* machines, and dynamically provisioning them is challenging in IoT[3] due to the heterogeneity of devices and their scattered ownership. Predicting resource utilization is also more difficult than in the *cloud* because many IoT applications are interfacing with the external environment, making them highly non-deterministic.

Memory-based process migration is unsuitable for IoT. Due to the aforementioned constraints of IoT devices, a great deal of flexibility is required in scheduling the workloads. In other words, static deployment of IoT applications to devices is insufficient, and hence there is a need for applications to be migratable. For example, an increase in the computational load or a decrease in battery level of a given IoT device might require delay-sensitive applications to be migrated to a different device. We may also need to migrate processes when there are external factors causing device failures (e.g., device gets overheated or physically damaged), or in the case of cyber-attacks on IoT devices.

In many dependability techniques, process migration is an important building block. At a high-level, it involves capturing the state of a running process, transferring a *snapshot* of the process (i.e., an *image* of the process state), then restoring the process from the snapshot. The same technique can be used to provide high availability and high resilience to crashes. The most intuitive and well-known process migration technique is the memory snapshot migration. In this technique, we simply copy the memory region of the running process to another device that has the same architecture. Unfortunately, such techniques are unsuitable in IoT due to the high heterogeneity of devices. Different devices have different memory layouts and instruction set architectures, and hence the snapshot serialization and deserialization steps will have to account for a variety of migration targets. Therefore, to be able to migrate a process from a device to any arbitrary device, the technique needs to be platform-independent.

1.2 | Our Work

Platform-Independent Migration of JavaScript Applications. We overcome the limitations of existing migration techniques by proposing ThingsMigrate[20], a comprehensive middleware for enabling platform-independent migration of stateful JavaScript applications across arbitrary IoT devices. ThingsMigrate automatically instruments the source code of a given program, injecting snippets of code into specific locations in the user program to access its internal state. The injected objects enable the system to capture the logical state of the user program on demand (e.g., via network request or an API call) and produce a platform-agnostic snapshot. Then, on the target device, new code is generated from the snapshot, which resumes the original program. No user intervention is required in the migration procedure and the interface is completely transparent to the application.

Other work has attempted to migrate browser-based applications. However, they either do not fully address some important JavaScript features – such as nested closures ([21]) – or they rely on VM-instrumentation[22], thereby making their approach dependent on a specific VM/browser implementation.

To the best of our knowledge, ThingsMigrate is the first comprehensive high-level framework for migrating stateful JavaScript IoT applications transparently without any user intervention, and without requiring any modifications to the JavaScript VM, providing platform-independent migration.

In summary, this paper provides the following contributions:

- A comprehensive JavaScript migration approach (Section 5) that is based on high-level code instrumentation and dynamic code generation, and that does not require VM modification, thereby allowing cross-platform migrations of JavaScript-based IoT applications.
- System implementation (Section 5.4) that handles many advanced features of the language and environment, such as arbitrarily nested closures, event queues, timers and MQTT-based communication interfaces, and support for multiple migrations.
- Optimizing the above migration technique, by using callback functions to lazily capture state, further improving the responsiveness and memory efficiency of the migration technique.
- Further optimization of the migration technique to mitigate the negative effects of maintaining explicit references to closure scopes.
- Evaluation through the execution of benchmarks across IoT and *cloud*-based devices (Section 6). Results indicate that ThingsMigrate can instrument arbitrary JavaScript programs, serialize their state and reconstruct them within microseconds and with average memory overhead of 90%. Further, ThingsMigrate supports multiple subsequent migrations with minimal memory usage increase.
- A case study (Section 7) which describes the experience of applying our approach in a real-world IoT context (motion detection over a video stream), predominantly using third-party libraries developed for server applications.

1.3 | Outline

The rest of this paper is organized as follows:

- **Challenges (Section 2)** discusses the challenges in migrating a JavaScript process during run-time.
- **Related Work (Section 3)** describes prior research in process migration in general, and similar work in JavaScript process migration.
- **Preliminaries (Section 4)** provides an overview of our migration technique, the assumptions we make about the application domain, a summary of the system architecture, and a formal description of the problem we address.
- **Approach (Section 5)** elaborates on the three different versions of our migration technique, its limitations, and the implementation details.
- **Experimental Validation (Section 6)** describes the experiments we conduct to evaluate our technique and discusses the observations.
- **Case Study (Section 7)** presents a case study of using our technique in migrating a video processing application.
- **Discussion (Section 8)** discusses additional thoughts related to our work, including the equivalency semantics of the three different approaches and the security impact of our technique.
- **Conclusion and Future Work (Section 9)** concludes this paper by providing a summary of our work and discussing future research directions.

2 | CHALLENGES

Migrating a running JavaScript program from one VM to another VM running on a different platform architecture poses several challenges when it comes to capturing and reconstructing the state. To support migration, the current state of the running process must be captured. As mentioned in Section 1.1, naively dumping the process memory into a byte array and replicating it on another machine would not work in this heterogeneous setting. Instead, we want to capture the *abstract logical state* of a running process. More concretely, in the context of JavaScript, this involves capturing the various objects bound to variables, function definitions, and – most importantly – closures and their context hierarchy. Once we capture everything needed to represent the entire process state, we need to serialize it into a platform-agnostic snapshot, and then restore the logical state entirely from the snapshot. We have identified the following challenges in doing so.

(1) Closures. In JavaScript, extracting an object and its properties is easy, using the standard JSON API (e.g., `JSON.stringify(foo)` where `foo` is an object). However, there are 2 critical limitations with this approach. The first is in correctly serializing the *static* definition of a function. The JSON schema[23] does not specify a serialization format for functions, and the onus is on the user to decide how functions should be serialized. Serializing a function is more involved than serializing the name of the function and the code in its body; we also need to serialize the *lexical context* surrounding the function. For instance, if a function refers to a variable `bar` that is outside its body, then we have to ensure that the variable `bar` exists in the right place when we restore the function so that we do not break the *lexical binding* of the said variable. The second challenge is in capturing the *dynamic state* of the closures created by function invocations and the local variables initialized inside these closures. This dynamic state is implicit and not visible at the code level. Accessing a closure's local scope is *impossible* through the application layer reflection APIs, as a function's local variables are not accessible from outside by definition. In fact, this semantics is fundamental in JavaScript and is what makes closures particularly useful; users can create a self-contained, stateful function that carries a private execution context. Since the state of closures is fundamentally *hidden* and cannot be accessed by an external agent, we need a mechanism to expose these hidden states and make them available for serialization.

(2) Object References. Since JavaScript is untyped and lacks a syntactic representation of *object references* (e.g., pointers), we cannot determine statically whether a given variable holds a literal value or a reference. It is important to distinguish between literal values and references because naively copying references into the snapshot would lead to duplication of objects if there are multiple references pointing to the same object. For example, consider an object `foo` with a property `bar` storing a circular reference to itself. If we do not check whether `foo.bar` points to `foo` itself, we would be recursively copying `foo` into itself and the snapshot would grow infinitely. Therefore, the links between various objects and their equality have to be assessed dynamically and without having access to the internals of the VM.

(3) Event Queue. JavaScript's execution model is based on an asynchronous event-loop, which repeatedly dequeues handler functions from the event queue in a FIFO fashion and calls them one after another until the event queue is exhausted. Although there are built-in APIs to interact with the event queue (e.g., `process.nextTick`), there is no way to inspect the event queue itself. Hence, this hidden state of the event queue and the associated event data needs to be exposed in order to restore the control-flow state of the program. The types of event handlers that need to be captured can be broadly categorized into: (1) timer events, (2) network events, and (3) file system events. We do not consider the Document Object Model (DOM) and user input events as typical IoT applications are running on server-side JavaScript VMs, and not in a browser; therefore, they do not have a DOM.

(4) Migration Trigger. As JavaScript is single-threaded, there is no easy way to interrupt the current execution (i.e., yield control mid-execution) at arbitrary points in time to perform a migration. Furthermore, the call stack is completely invisible to the JavaScript layer and cannot be accessed or modified without access to the VM's internals. Therefore, we need to come up with a mechanism to trigger the migration at certain yield points in the execution of the program.

(5) State Reconstruction. Assuming we have a way to capture the state of a program, restoring the program from a serialized image poses yet another challenge. A snapshot is a *static representation* of the *dynamic, run-time state* of a program. Given a static image, we have to restore the dynamic state of a program without the ability to directly manipulate the internal state of a VM – that is, statically through code generation. Restoring the dynamic state through code generation is non-trivial, as we have to preserve the hierarchy of closures and the implicit references between objects, and without re-executing code that can potentially lead to side effects.

(6) Multiple Migrations. As a given program might be migrated arbitrarily many times, we need to support multiple migrations. For this to work, the migration mechanism must ensure that the restored program is *semantically equivalent* – or at the very least *observationally equivalent* – to the program before snapshot. If this condition is not met, then the program behaviour may deviate over multiple hops of migration, which would be incorrect. To ensure that a restored program is equivalent to the original program, the snapshot and restore procedures have to satisfy the following:

Formal Definition. Let P be the set of all *program configurations* (dynamic state during run-time) and S be the set of all serialized process images. We define the snapshot function $snapshot : P \rightarrow S$ and the restore function $restore : S \rightarrow P$. Then, given a program $p_o \in P$, we have to implement *snapshot* and *restore* functions in such a way that: $restore(snapshot(p_o)) \Leftrightarrow p_r \Leftrightarrow p_o$ where p_r is the restored program. If we cannot satisfy the relation $p \Leftrightarrow restore(snapshot(p))$, the restored program may exhibit deviant and incorrect behaviour. In other words, *snapshot* and *restore* should both be bijective functions and thus inverse of each other. The migration technique must implement the corresponding *snapshot* and *restore* procedures to make sure that a given program can survive and remain functionally equivalent over multiple hops of migration.

3 | RELATED WORK

To be clear, we reiterate that our work is about migrating a live JavaScript process during run-time, and not about migrating the static JavaScript code from one platform to another. Hence, we consider related work in VM migration and process migration, but not those in file system migration, database migration, or more recently, blockchain migration[24].

Migration of JavaScript Applications. There has been significant prior work in the area of migrating JavaScript programs. However, they focus on migrating web applications between web browsers[21,22,25,26], and hence have different constraints and assumptions than in the context of IoT devices e.g., capturing the DOM state and user input. Consequently, some of the techniques[22,26] require modification to the underlying JavaScript VM to access the hidden application state, making them platform-dependent.

Imagen[21] migrates web applications across heterogeneous browsers without altering the VM, and address some of the challenges specific to web applications (e.g., the DOM, HTML5 media elements). However, their handling of nested closures is limited (Section 5.1.1). Further, Imagen allows the application to be migrated just once, as it restores the scope hierarchy by creating a global dictionary object and the restored code lacks the necessary instrumentation for subsequent migration. The restored program thus may exhibit the same behavior after a single hop of migration, but its lexical structure is altered, breaking semantic equivalence.

	ThingsMigrate	Imagen[21]	Kwon et al.[22]	FlashFreeze[27]
Application Domain	**Server**	Browser	Browser	Server
Supports DOM Migration	**X**	O	O	X
Supports Migrating Deep Closures	**O**	X	O	O
Supports Multi-hop Migration	**O**	X	X	X
Is Platform-independent	**O**	O	X	O

TABLE 1 Comparison between JavaScript migration techniques. X means that the technique does not support a feature, while O means it does.

Similar to Imagen, Oh et al.[26] propose a migration framework for web applications, with limited support for capturing closure states. Kwon et. al.[22] further extend their work to provide deeper support for serializing and reconstructing nested closures, though they report a degradation in performance with growing closure depth. Furthermore, they require modifications to the JavaScript VM to access the internal scope hierarchy, which makes their approach less portable and tied to a specific version of the open-source Webkit browser.

FlashFreeze[27], which claims to be inspired by our previous work[20], demonstrates a more performant process migration also based on code instrumentation. The performance boost comes mostly from capturing the program state lazily. In the code instrumentation step, they first statically extract the names of the variables captured by a closure, and inject a *capture list generator* function for each closure, which they invoke at the time of snapshot to retrieve the captured variables. Similar to Imagen[21], FlashFreeze cannot subsequently migrate a program after restoring it from a snapshot, making it unsuitable for our target application domain where a program needs to be migrated multiple times.

Table 1 summarizes how prior work in the migration of JavaScript programs compares against ThingsMigrate in terms of the features supported by each technique or framework. As shown in the table, ThingsMigrate is the only work that supports multiple hops of migration (i.e., repeated migration), since it preserves the program semantics between migration. ThingsMigrate, FlashFreeze, and Imagen are platform-independent as they are based on code-instrumentation; though it should be noted that Imagen is actually implemented by extending Mozilla Rhino[28] – a JavaScript VM written in Java – rather than as an integrated JavaScript module. Kwon et. al.[22] requires access to the underlying JavaScript VM and thus is not platform-independent.

Deterministic Replay. As an alternative to capturing and restoring the state of the web application, deterministic replay techniques can be used to replay an exact sequence of actions leading to the current state[29,30,31,32,33]. However, these approaches focus on web browser events, and are hence not applicable to IoT environments. Further, they are not practical for IoT environments that are resource-constrained, as the sequence of events to be captured and replayed grows rapidly over time[21].

VM/Container Migration. As mentioned in Section 1.1, there has been many attempts at providing low-level migration techniques that directly save and restore the process memory space, and are hence programming language independent[34,35,36,37,38]. Recent work in process migration in the mobile/edge/cloud environments[39,40,41] also use a memory-snapshot based technique. Such techniques could be applied for migrating JavaScript programs, but they would require serializing the state of the JavaScript virtual machine (VM) itself, which can incur significant overheads on IoT devices. Further, as a single VM might host several IoT components, migrating the entire VM would migrate all the components. Most importantly, providing platform-independent migration would not be possible in diverse IoT environments, as even the same version of a given VM might have different in-memory representations across platforms due to hardware differences. Similar challenges arise in migrating virtualized OSes across devices[42,43], illustrating the limitations of migration techniques based on memory-snapshots. We avoid the challenges posed by platform differences by leveraging the platform-independent environment provided by the language runtime (in our case, JavaScript), and by employing code instrumentation in the application layer.

4 | PRELIMINARIES

We first present the overall workflow at a high-level, and then detail the assumptions we make, followed by the system model. We then present a motivating example to illustrate the migration process, followed by the problem statement. In Section 5, we present 3 different approaches for performing the migration to solve the problem as per this workflow.

4.1 | Overview

The migration process consists of 3 steps:

(1) Code Instrumentation: this step is performed prior to running a given user application, and only needs to be done once for each application. The main purpose of this step is to expose the hidden states in the program by injecting certain ThingsMigrate objects into the appropriate locations, which will have access to the hidden objects. Once the code has been instrumented, we can then access the internal states of the program through the injected ThingsMigrate objects. During this step, we also inject an event listener that listens for snapshot requests, providing an interface to interact with the program. (Section 5.1.1)

(2) State Extraction and Serialization: this step is triggered by an external entity (i.e., end-user or ThingsMigrate Manager) through the event listener interface injected during the code instrumentation step. The instrumented application provides access to all the objects in the program, but these objects are still in their native form inside the VM. To create a snapshot that can be transported, we have to serialize the objects, the relations between them, and the hierarchy of the scopes in which different objects reside. We do this by recursively traversing the tree of ThingsMigrate `Scope` objects (injected in step (1)), serializing each node along the way, and finally producing a snapshot that represents the tree-like logical state of the program. This snapshot is then transported to another device where the next step happens (Section 5.1.2).

(3) Code Reconstruction: given a platform-agnostic snapshot, we generate a new program that is equivalent to the program at the time the snapshot was taken. The reason we have to generate a new program is that we cannot directly control the VM to create the implicit closure states. Instead, we restore the closure scopes by writing *immediately invoked functions* in the newly generated program. We reconstruct the program in such a way that the overall logical structure of the program is unchanged, and only the data and control flow states are updated; this is crucial for enabling subsequent migrations. The generated program is then executed as if it were a freshly instrumented program (Section 5.1.3).

Note that steps (1) and (3) can occur anywhere, and not necessarily on the IoT device. In fact, the initial device (where the application is initially run) can perform all 3 steps and the restored code itself can be transmitted to another device. However, for efficiency, we perform the instrumentation on a high-performance machine e.g., *cloud* instance. Code reconstruction is done by the target device on which the application will be restored. This choice is made for pragmatic reasons.

4.2 | Assumptions

We make five assumptions as follows.

ES5 Compliance. To ensure broadest compatibility, we assume that the JavaScript code is compliant with strict-mode ES5 (ECMAScript 2009)[44], which has been the *de facto* standard for many years. Although more recent versions of JavaScript have been released and are now widely adopted (e.g., ES6/ECMAScript 2015[45]), not all JavaScript engines fully support the newest features of the language. Supporting programs that use language constructs from newer versions of the ECMAScript standard is not a significant issue, as support for ES6+ can easily be provided by leveraging transpilers (e.g., Babel.js[46]) to convert to ES5.

Asynchronous Programming Best Practices. Because JavaScript is event-driven and single-threaded, we assume that developers will avoid blocking the main thread for long periods of time, as this would prevent the migration from being scheduled. Note that this assumption is not specific to ThingsMigrate – in fact, a long-running operation that never yields control would inhibit any asynchronous event (e.g., timers, messages, I/O) from being dispatched. To use ThingsMigrate, developers should follow the best practices and write their code in an event-driven manner, or break long-running operations to yield control (e.g., `setImmediate`) at periodic intervals.

Use of Publish/Subscribe. We assume that applications use the MQTT.js Publish/Subscribe (Pub/Sub) API for network communication, instead of the built-in `net` or `http` module. Since Pub/Sub is the de-facto communication interface in most IoT applications[47], we believe this assumption is reasonable. That said, our technique is not specific to the Pub/Sub interface, and can easily be adapted to other mechanisms.

Use of Local File System. We also assume that applications do not perform write operations to the local file system. The migration of applications writing to local files is challenging. For instance, we have to answer the question of whether to migrate the file itself to the target device, so that when the application resumes, it is writing to the same file. Alternatively if we decide not to migrate the file, we have to address how to handle consistency of files distributed across different devices. In this paper, we only focus on high-level process migration mechanism, and leave these decisions for future work (Section 5.1.4).

Migration support in the VM. While migration support could be implemented in the VM, we believe that this is unlikely in the near future given the vast heterogeneity in IoT platforms. Unlike in the web browser space where there are only a handful of dominant players, the IoT landscape is much more fragmented, with the availability of a wide variety of JavaScript engines (e.g.,

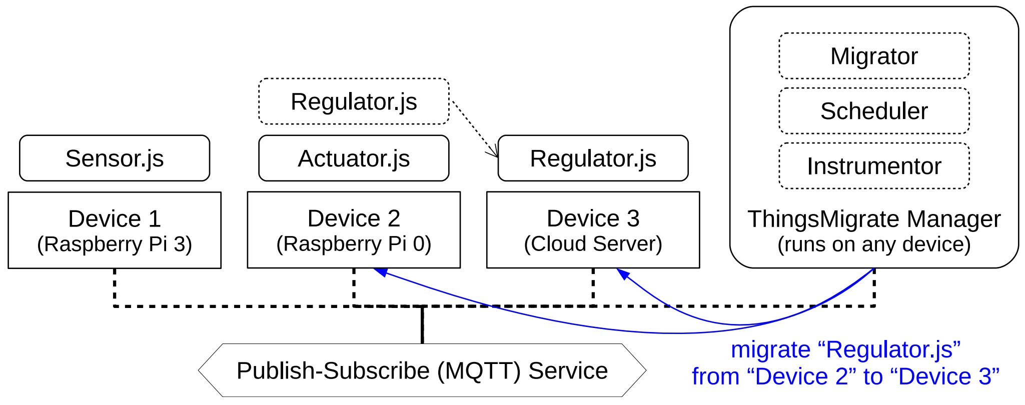


FIGURE 1 Schematic diagram of ThingsMigrate system architecture consisting of 3 components – (1) Instrumentor instruments each JavaScript program initially to make them migratable, (2) Scheduler determines the appropriate migration target, and the (3) Migrator coordinates the migration process by issuing commands over the Publish-Subscribe network. Blue lines refer to the migration operations.

[5,6,12,13]). Further, migration support would be required at both ends of the migration process; i.e., at the source device/VM, and at the target device/VM, which may be different from each other. Some of the VMs may be closed-source and hence not easily modifiable. That being said, should full or partial support for migration be provided in the VM (e.g., by enabling special APIs to access the state of closures), our technique could be adapted accordingly.

4.3 | System Model

To describe the overall workflow and provide some context in which our migration technique would be used, we first present the underlying system architecture of ThingsMigrate, as illustrated in Figure 1. The system is derived from the architecture of ThingsJS, a comprehensive IoT middleware that we presented as a vision paper in[8]. *While ThingsJS proposes migration as part of an integrated system, it does not specifically address migration challenges.* We describe the systems components below.

4.3.1 | ThingsMigrate Manager

The central piece of our architecture, namely the *ThingsMigrate Manager*, manages the execution of distributed IoT applications across the set of available devices. In our model, all communications between the components of the system use the topic-based Pub/Sub paradigm (MQTT)[48]. Pub/Sub decouples content producers (*publishers*) from content consumers (*subscribers*), providing a higher-level of abstraction free from the low-level network considerations.

Overall, the *Manager* has three components:

(1) Scheduler. This component orchestrates the execution of IoT applictions across all devices. For the *Scheduler* to operate efficiently, developers are encouraged to modularize their IoT applications into a set of *Components*, and to follow the best practices of JavaScript (Section 4.2). Taking into consideration the capabilities of each device, the requirements of the *Components*, and a set of developer-specified constraints, the *Scheduler* determines the assignment of each *Component* onto a specific device. Upon changing conditions, the *Scheduler* can decide to dynamically migrate some of the *Components* between devices. In this paper, we assume that the *Scheduler* is responsible for initiating a migration; the details of the *Scheduler* (e.g., scheduling algorithm, etc.) are outside the scope of this paper.

(2) Instrumentor. This component is in charge of instrumenting the JavaScript source code of the IoT applications, which is eventually executed by the *ThingsMigrate Runtime*. The code instrumentation procedure is performed once before running an application on ThingsMigrate. Most of the apparatus needed to enable migration is bootstrapped at the code instrumentation stage, and we discuss the work done by the *Instrumentor* in depth in the following sections – this is our main contribution.

(3) Migrator. The *Migrator* is in charge of transparently migrating the running application from a source device to the destination device. To migrate a given *Component* (e.g., *Component regulator1* on device 2), the migrator issues a migrate command to the *ThingsMigrate Runtime* running on the source device (Section 4.3.2), specifying the name of the *Component*

```
 1 function makeAccount(name, initial){
 2    var _balance = initial || 0;
 3    return {
 4       name: name,
 5       balance: function(amount){
 6          if (typeof amount === 'number') _balance += amount;
 7          return _balance;
 8       },
 9       makeTransfer: function(account, amount, repeat){
10          var _count = 0;
11          return function installment(){
12             if (_count < repeat){
13                _balance -= amount;
14                account.balance(amount);
15                _count ++;
16             }
17             console.log(name+' $' + _balance + ', ' + account.name + ' $' + account.balance());
18          }
19       }
20    }
21 }
22 var alice = makeAccount('Alice', 100);
23 var bob = makeAccount('Bob');
24 var transferOut = alice.makeTransfer(bob, 20, 4);
25 var transferIn = alice.makeTransfer(bob, -10, 6);
26 setInterval(transferOut, 1000);
27 setInterval(transferIn, 1500);
```

FIGURE 2 Plain JavaScript Code Example

to migrate. The source *Runtime* returns a snapshot back to the *Migrator*, which then issues a restore command to the destination *Runtime*, passing the snapshot with the command. We only discuss the Migrator's role in passing in this paper.

4.3.2 | ThingsMigrate Runtime

The *ThingsMigrate Runtime* is a thin JavaScript middleware service that runs on each IoT device and manages the local execution of all the *Components* running on the device. It receives the instrumented source code of various *Components* from the *Instrumentor* and executes them, and awaits migration commands from the *Migrator* over the Pub/Sub interface. Upon receiving a *migrate* command for a *Component* (e.g., for *Component regulator1* on device 2), the Runtime first captures the state of the running program through the ThingsMigrate API injected into the augmented program, then serializes its state (Section 5.1.2) and sends it back to the *Migrator* over Pub/Sub. Alternatively, when a Runtime (e.g., destination device *device1*) receives the serialized snapshot of a *Component*, it restores the program by generating the appropriate restoration code (Section 5.1.3), which resumes from the pre-migration state.

4.4 | Motivating Example

Figure 2 presents an example JavaScript program that we use throughout this paper as a running example. In lines 1-21, there is a Function Declaration defining `makeAccount`, which returns an object with 2 properties – `balance` and `makeTransfer` – each referencing an anonymous function. The 2 anonymous functions (line 5, 9) access the variable `_balance` declared in the local scope of `makeAccount`, and continue to have access to it after `makeAccount` has *returned*; thus, both functions are closures. The closure function `balance` in line 5 is used as both *getter* and *setter* for the "private" variable `_balance`. It takes a single argument, adds it to the closed variable `_balance` if it is a number, and returns the new `_balance`. If the argument is not a number, it simply returns `_balance`. The `makeTransfer` closure in line 9 is slightly more complex. It creates a local variable `_count` and then returns another closure function `installment` (line 11) that captures it. The variable `_count` is used to keep track of how many times `installment` was called. `installment` is used to decrement a given `amount` from the closed variable `_balance` and add the same `amount` to the given `account` - it can be invoked up to `repeat` number of times. In line 22, a new variable `alice` is declared and initialized by invoking `makeAccount`, which returns an object containing the 2 closures (`balance` and `makeTransfer`). The hidden variable `_balance` is initialized to 100. While the function `makeAccount` has *returned*, its local variable `_balance` is still "in-scope" because of the 2 closures referencing it. We simply refer to this

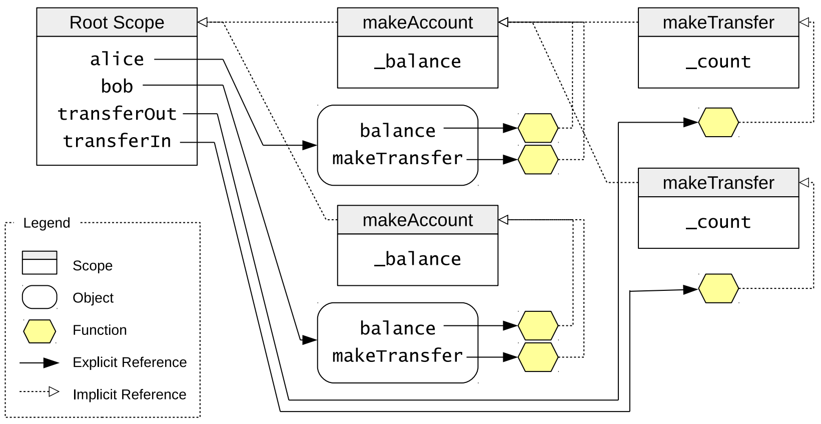


FIGURE 3 JavaScript Example Closures and Scopes

hidden scope containing the `_balance` as `alice`'s scope from now on. In line 23, a similar object is created and assigned to the variable `bob`, with a `_balance` of 0. In line 24, `alice.makeTransfer` is called to create an `installment` function, which can be called up to 4 times, each time transferring an `amount` of 20 from `alice`'s `_balance` to `bob`'s `_balance`. This `installment` closure is assigned to the variable `transferOut`. Similarly in line 25, a different `installment` closure is created and assigned to the variable `transferIn`. Finally in line 26 and 27, the native `setInterval` API is used to schedule the `transferOut` and `transferIn` function to be called every 1 second and 1.5 seconds respectively.

Figure 3 visually illustrates the *scopes* of the various closures and their relationship. There are two independent copies of variable `_count` each defined in their own scope, but only one copy of `_balance`, which is defined in the parent scope and is hence *shared* with the two child scopes.

4.5 | Problem Statement

ThingsMigrate captures the hierarchy of *scopes*, starting from the global scope, as well the data elements (variables and functions) contained within each scope. In other words, ThingsMigrate captures the *structure* and the *values* of the different state elements. More formally, we denote the state of a JavaScript application as $\mathbb{S} = \langle S, F, V, R\rangle$, where S is the set of *scopes*, F is the set of *functions*, V is the set of *variables* (i.e., a tuple of $\langle$name, value$\rangle$) and R is the set of *relations* between scopes and other entities (i.e., a tuple of $\langle$scope, entity$\rangle$).

Taking the code snippet shown in Figure 2 as an example, and assuming that a snapshot of the state is taken after 3250ms, there are 2 objects that need to be captured, each representing `alice` and `bob`. Each object contains 2 closures referencing a closed variable `_balance`. The scope containing `_balance` and its closure functions should be included in the snapshot. Also in the global scope are 2 instances of the `installment` function, each enclosing a variable `_count`. The `installment` closures were returned by the `makeTransfer` function inside `alice`'s scope. The resulting state of the snapshot object $\mathbb{S} = \langle S, F, V, R\rangle$ would respectively contain states S, functions and their definition F (omitted for brevity), variables and their value V, and the set of relationships R between each variable/function and its associated scope:

$$
\begin{aligned}
S =&\{\sigma0, \sigma1\_\texttt{alice}, \sigma1\_\texttt{bob}, \sigma3\_\texttt{t1}, \sigma3\_\texttt{t2}\}\\
F =&\{\texttt{makeAccount}, \lambda1\_\texttt{alice}, \lambda2\_\texttt{alice}, \lambda1\_\texttt{bob}, \lambda2\_\texttt{bob}, \texttt{installment\_t1}, \texttt{installment\_t2}\}\\
V =&\{(\texttt{alice}, \texttt{<Object>}), (\texttt{bob}, \texttt{<Object>}), (\texttt{transferOut}, \texttt{installment\_t1}), (\texttt{transferIn}, \texttt{installment\_t2})\},\\
&\{(\texttt{\_balance\_alice}, 60), (\texttt{\_balance\_bob}, 40), (\texttt{\_count\_t1}, 3), (\texttt{\_count\_t2}, 2)\},\\
R =&\{(\sigma0, \texttt{makeAccount}), (\sigma0, \texttt{alice}), (\sigma0, \texttt{bob}), (\sigma0, \texttt{transferOut}), (\sigma0, \texttt{transferIn})\},\\
&\{(\sigma1\_\texttt{alice}, \lambda1\_\texttt{alice}), (\sigma1\_\texttt{alice}, \lambda2\_\texttt{alice}), (\sigma1\_\texttt{alice}, \texttt{\_balance\_alice})\},\\
&\{(\sigma1\_\texttt{bob}, \lambda1\_\texttt{bob}), (\sigma1\_\texttt{bob}, \lambda2\_\texttt{bob}), (\sigma1\_\texttt{bob}, \texttt{\_balance\_bob})\},\\
&\{(\sigma3\_\texttt{t1}, \texttt{installment\_t1}), (\sigma3\_\texttt{t1}, \texttt{\_count\_t1}), (\sigma3\_\texttt{t2}, \texttt{installment\_t2}), (\sigma3\_\texttt{t2}, \texttt{\_count\_t2})\}
\end{aligned}
$$

TABLE 2 AST Node Reference Table

Abbreviation	Name	State-changing	Example
IDNT	Identifier	×	`foo`
OBJEXPR	Object Expression	×	`{ foo: 1 }`
FUNCEXPR	Function Expression	×	`function(){ }`
MEMBEXPR	Membership Expression	×	`foo.bar`
UNAEXPR	Unary Expression	×	`!foo`
BINEXPR	Binary Expression	×	`foo > bar`
LOGICEXPR	Logical Expression	×	`foo && bar`
UPDTEXPR	Update Expression	○	`foo ++`
ASSGNEXPR	Assignment Expression	○	`foo = 1`
FUNCDECL	Function Declaration	○	`function foo(){ }`
VARDECL	Variable Declaration	○	`var foo = 1`
CALLEXPR	Call Expression	△	`foo()`
NEWEXPR	New Expression	△	`new Foo()`
RTNSTMT	Return Statement	△	`return foo`

×: Does not change state
○: Explicitly updates state
△: Implicitly updates state via scope creation/destruction

Section 5.1.3 (Code Restoration) describes in more detail our algorithmic approach to generating reconstruction code, and gives an example of the restored code of the same code sample (Figure 2) after migration. As can be observed, the same functions, scopes and variables, as well as their relationships, are in the restored code sample (Figure 6).

For ease of discussion, we briefly introduce some terminology relating to the JavaScript Abstract Syntax Tree (AST). Table 2 lists the AST nodes relevant to ThingsMigrate, and we will use the abbreviations in column 1 to refer to a certain type of expression. Column 3 indicates whether the given expression has an effect on the program state during run-time. Expressions like MEMBEXPR do not alter the state, so we refer to them as *no-change* expressions. On the other hand, expressions like ASSGNEXPR directly modify the values of variables, and we refer to them as *explicit-change* expressions. CALLEXPR and NEWEXPR are more interesting, since they do not directly update any state variable. However, they create new scopes containing new variables and change the structure of the internal scope tree; we refer to these nodes as *implicit-change* expressions.

The next sections describe the algorithmic process followed by ThingsMigrate to (1) instrument the code to expose the hidden states, (2) take a snapshot and (3) reconstruct the code at the serialized state. As mentioned earlier, we consider three different techniques to perform the above operations with different tradeoffs in terms of memory and performance.

5 | APPROACH

In this section, we discuss 3 approaches for performing JavaScript process migration. We first present our initial approach to migrating a JavaScript process during run-time, which we label TREECOPY. Based on the lessons gained in the first approach, we present 2 additional approaches – XPLICTGC and LAZYSNAP – that attempt to optimize the performance overhead of migration. Finally, we present the details our our implementation.

5.1 | Technique 1: "TREECOPY"

5.1.1 | Code Instrumentation

In the code instrumentation phase, the ThingsMigrate Runtime augments the input JavaScript source file to allow the state to be dynamically captured (challenge 1), corresponding to the formal model defined in Section 4.5. Our code instrumentation approach is inspired by the work by Lo et. al.[21], but differs significantly as Lo et. al.[21] only offers limited support for capturing and restoring complex closures. In the example shown in Figure 2, Lo et. al.[21] would be able to capture and restore the scope of

```
Scope {
    creator: Function               // the function that created this scope
    params: [ (name, value) ]       // tuples of parameters the creator was invoked with
    uid: String                     // unique id of the scope
    parent: Scope                   // the parent scope
    vars: [ (name, value), ... ]    // local variables defined in the scope
    funcs: [ Function, ... ]        // functions created in this scope
    children: [ Scope, ... ]        // child scopes
}
```

FIGURE 4 Scope Object Definition

the two internal *installment* closures, but would not accurately model the relationship between said scopes in the restored output, so that two different instances of the `makeAccount` scope would be generated rather than one, ending with two distinct `_balance` variables after restoration. Each nested scope would then update its own `_balance` variable, which would be inaccurate.

The main aspects of our technique are illustrated in Algorithm 1. The high-level idea behind this approach is to expose the internals of the logical data structure by injecting additional objects, so that the instrumented program maintains an explicit copy of the data used by the program during run-time. In order to fully capture the state of closures, the *Instrumentor* exposes the hidden variables by injecting `Scope` objects into every function, which will store the variables defined locally in its scope, and then dynamically constructing a *scope tree* that mirrors the internal scope hierarchy. An abstract representation of the `Scope` object is presented in Figure 4.

```
1  function instrument(code: String) : String
2      ASTroot ← codeToTree(code)
3      ASTroot ← instrumentNode(ASTroot)
4      newCode ← treeToCode(ASTroot)
5      return wrapTemplate(newCode)
6  end
7  function instrumentNode(node: ASTNode, parentScope: LexicalScope) : void
8      switch node.type do
9          case FuncDecl or FuncExpr do
10             scope ← new LexicalScope(parentScope)
11             foreach childNode in node.body do
12                 injection ← instrumentNode(childNode, scope)
13                 if injection then
14                     node.body.insertAfter(childNode, injection)
15                 end
16             end
17             firstLine ← new ASTNode("var " + scope.name + " = new Scope(" + parentScope.name + ")")
18             node.body.unshift(firstLine)
19             injectAfter ← new ASTNode(parentScope.name + ".addFunction(" + node.name + ")")
20             return injectAfter
21         case VarDecl do
22             parentScope.addVariable(node.name, node.value)
23             injectAfter ← new ASTNode(parentScope.name + ".vars." + node.name + " = " + node.name)
24             return injectAfter
25         case AssgnExpr do
26             varScope ← parentScope.findVariableScope(node.name)
27             injectAfter ← new ASTNode(varScope.name + ".vars." + node.name + " = " + node.name)
28             return injectAfter
29         otherwise do
               /* If node has child nodes, call instrumentNode recursively.    */
               /* Otherwise return.                                            */
30         end
31     end
32 end
33 function wrapTemplate(code: String) : String
34     return "require('things-js').bootstrap(function(){" + code + "})"
35 end
```

Algorithm 1: Code Instrumentation Algorithm (V1)

Lines 1-6 in Algorithm 1 describe the user-facing API for performing code instrumentation. Note that an end-user does not need to explicitly call this function, as the ThingsMigrate Runtime automatically invokes it before running the program given by the user. The `instrument` function accepts the code string as its only argument, and returns the code string of the instrumented program. The first step in instrumentation is to construct an Abstract Syntax Tree (AST). We then pass the root node of the AST to the function `instrumentNode`, which will traverse the AST recursively and update each node as needed. Lines 7-32 illustrate the important aspects of the `instrumentNode` procedure, omitting some of the minor details for clarity. The kind of operation performed on a node depends on the type of the AST node. If the node currently being processed is a function (i.e., FUNCDECL or FUNCEXPR), we need to inject a `Scope` object because a function invocation will implicitly create a new scope, which we need to expose. For example, when the `makeAccount` function in Figure 2 is called, a hidden scope will enclose the variable `_balance` for that very invocation of the function. In line 10, we first create a new `LexicalScope` object in order to keep track of the variables and functions being defined in its scope. It should be noted that the `LexicalScope` object only tracks the lexical information during instrumentation and is different from the `Scope` object in Figure 4, which represents the dynamic scope of a function during run-time. After instantiating a `LexicalScope` object, the algorithm proceeds to instrument the function body. We iterate through the nodes in the function body (lines 11-16) and invoke `instrumentNode` on each AST node recursively, some of which return an *injection*. The *injection* is a piece of code that is placed immediately after the expression that was instrumented (line 14). After the body of the function is processed, we inject a line of code that instantiates the appropriate `Scope` object at the beginning of the function being instrumented (lines 17-18). Finally, since the function being instrumented is itself an object belonging to the parent scope, we create a line of code that will add the function to its parent scope (line 19), and then return it (line 20) as an *injection*.

Instrumenting a VARDECL node is more straightforward (lines 21-24). We first register the variable with the `LexicalScope` object (line 22) so that any other nested nodes referencing the variable can find its enclosing scope. We then inject a line of code that will update the variable's value in the corresponding `Scope` object (lines 23-24). When processing an ASSGNEXPR, we first look up the lexical scope in which the variable was declared by traversing the chain of `LexicalScope` objects until the variable is found (line 26). Subsequently a line of code is injected that will update the variable's value in the corresponding `Scope` object. There are several other types of JavaScript statements that are traversed, but we skip the operational details as they are immaterial to the instrumentation technique. Once the `instrumentNode` called on the root node returns, the AST represents the instrumented version of the program (line 3). We convert the AST back to code (line 4), and then finally return it to the user-space after wrapping the entire application code in a ThingsMigrate template code (line 5). The template is a small snippet of *glue code* that performs *bootstrapping* work such as importing the ThingsMigrate objects (e.g. `Scope`) and connecting to the Pub/Sub service to listen for incoming snapshot commands.

Figure 5 depicts the instrumented version of the original source code shown in Figure 2. The lines of code that are added during the code instrumentation process are shown in bold. As can be observed, all defined scopes (the global scope, then the scopes corresponding to each function definition) are mirrored through an instance of a ThingsMigrate `Scope` object (lines 1, 3, 9, 17, and 21). Furthermore, each variable is copied into the `Scope` at which it is defined (lines 5, 12, 19, 24, 27, 36, 38, 40, and 42). Similarly, functions are also registered (lines 8, 16, 20, and 34). To make it easy to identify the anonymous functions, the *Instrumentor* simply assigns a unique name to each anonymous function (lines 8, 16).

Instrumenting Timers. Following a similar algorithmic approach as Lo et. al.[21], ThingsMigrate provides support for saving the state of timer functions, namely `setInterval` and `setTimeout` (challenge 3). This is accomplished in the instrumentation phase by replacing the native timer calls with the ThingsMigrate timer API (lines 43, 44 in Figure 5), which expose the state of the timers such as remaining time until the next invocation of the callback. Consequently, the restored timers resume from the state it was in when serialization took place. For instance, if we took a snapshot of our example program after 3250ms, then upon restoration, the first timer (line 43) will trigger its callback after 750ms and then every second, while the second timer (line 44) will trigger after 1250ms and then every 1500ms.

Pub/Sub Interfaces. ThingsMigrate provides support for capturing the state of Pub/Sub interfaces (challenge 3), following our assumptions (Section 4.2) about the usage of Pub/Sub in IoT. Similar to how we handle timers, ThingsMigrate wraps calls to the Pub/Sub interface (MQTT library) at the instrumentation phase, so that upon a migration being requested, the *list of each topic previously subscribed* by the application gets serialized as part of the snapshot. Then, at the restoration phase, prior to resuming the execution, a subscription is transparently reestablished to each of the previously subscribed topics. To ensure that no publications are lost *during the migration*, we assume that reliable Pub/Sub is provided by the service[49,50], so that the latter can retransmit any missed publication sent during the migration.

```
var σ0 = new Scope();
function makeAccount(name, initial){
   var σ1 = new Scope(σ0);
   var _balance = initial || 0;
   σ1.vars._balance = _balance;
   return {
      name: name,
      balance: σ1.addFunction(function λ1(amount){
         var σ2 = new Scope(σ1);
         if (typeof amount === 'number'){
            _balance += amount;
            σ1.vars._balance = _balance;
         }
         return _balance;
      }),
      makeTransfer: σ1.addFunction(function λ2(account, amount, repeat){
         var σ3 = new Scope(σ1);
         var _count = 0;
         σ3.vars._count = _count;
         return σ3.addFunction(function installment(){
            var σ4 = new Scope(σ3);
            if (_count < repeat){
               _balance -= amount;
               σ1.vars._balance = _balance;
               account.balance(amount);
               _count ++;
               σ3.vars._count = _count;
            }
            console.log(name+' $' + _balance + ', ' + account.name + ' $' + account.balance());
         })
      })
   }
}
σ0.addFunction(makeAccount);
var alice = makeAccount('Alice', 100);
σ0.vars.alice = alice;
var bob = makeAccount('Bob');
σ0.vars.bob = bob;
var transferOut = alice.makeTransfer(bob, 20, 4);
σ0.vars.transferOut = transferOut;
var transferIn = alice.makeTransfer(bob, -10, 6);
σ0.vars.transferIn = transferIn;
σ0.setInterval(transferOut, 1000);
σ0.setInterval(transferIn, 1500);
```

FIGURE 5 JavaScript Code Example - instrumented with V1

Classes and Prototypes. The JavaScript language does not support classes *per se* unlike object-oriented languages (e.g., Java). Instead, it provides high-level abstractions that emulate classes by means of *prototypal inheritance*[51]. ThingsMigrate provides support for serializing JavaScript-like classes by serializing each object's *prototype object*, so that upon restoring the code, the correct prototype chain can be recreated along with the objects.

Cleaning Orphaned Scopes. During the life cycle of a JavaScript application, scopes are dynamically created, and can sometimes become *orphaned*. Orphaned scopes are scopes for which there are no single remaining references that point to either them or one of their child scopes. In the example shown in Figure 2, at each timer iteration (line 26), the scope that is created on the fly by the `transferOut` function (first argument) becomes orphaned and is therefore destroyed, as its serialization will not be required. Therefore, we need to manually destroy the scope objects corresponding to orphaned scopes, as they can lead to memory leaks otherwise – this problem is exacerbated on multiple migrations (challenge 6).

As a novel contribution, ThingsMigrate provides support for automatically destroying orphaned scopes, to support multiple migrations (challenge 6) on the same application without increasing the snapshot size and incurring additional overhead in the restored code (i.e., scope explosion). In the instrumentation phase, the argument of a RtnStmt is wrapped in an API call to `Scope.maybeDestroy` (line 16 of Algorithm 1), for the current `Scope` object. At execution time, this function will check whether any other `Scope` or variable depend on this `Scope`. If there are no dependent objects, then the scope is destroyed, and therefore it will not be included in the snapshot (Section 5.1.2).

5.1.2 | Snapshotting and Migrating

To trigger a migration, the component that is being executed receives a *migrate* command from the *Migrator Service* through the Pub/Sub interface. Recall that the code instrumentation phase sets up a listener, which initiates the migration (Section 5.1.1).

Serializing the State. The migration process first involves serializing the state to the JSON format. To do so, the scope tree is recursively walked in a top-down approach, from the global scope. The serialized output includes, for each scope, the variables and parameters, as well as nested scopes and functions. In JavaScript, functions cannot be serialized as-is. Thus, upon encountering a function when walking the scope tree, the function is assigned a unique ID, and the function's source code is added to a table of functions, which is appended at the end of the serialized state. Note that the serialized output also contains the state for special objects that ThingsMigrate addresses, such as timers and Pub/Sub interfaces.

Handling the Stack. We address the challenge of handling the stack (challenge 4) by exploiting the asynchronous, event-driven nature of JavaScript. Because JavaScript applications are single-threaded and are event-based, the runtime maintains an event queue. We *schedule* code migrations as events so that they get pushed at the end of the event queue and get executed over an *empty* stack. More precisely, as migration requests are sent through the form of Pub/Sub publications, they are treated as events and pushed to the event queue. We could also accomplish the same behavior by scheduling the migration as a timer-based event.

Sending the Serialized State. Once the snapshot is generated, it is sent over the Pub/Sub interface to the target IoT node, which will regenerate the code considering the state of the snapshot, and resume execution.

5.1.3 | Code Restoration

Upon a given IoT node receiving a snapshot, it needs to reconstruct the original program *at the exact state* where migration took place (challenge 5). The code restoration process must retain the original program structure, while reassigning the values of logical constructs holding state, such as variables, parameters and closures, without directly restoring the memory regions - this is important for platform independence and portability.

Reconstructing Closures and Scopes. As in the code instrumentation phase, closures pose unique challenges when it comes to generating restoration code, as they wrap state elements. Because functions can have return values as functions in JavaScript (e.g., as seen in line 11 in Figure 2), there can exist multiple instances of the same function, each with its own scope and maintaining different *states* (i.e., holding different values in closed variables). The code restoration process needs to instantiate multiple copies of the same function while preserving the hierarchy of the associated scopes, since closed variable states can not only be held in its enclosing scope but also anywhere in its ancestor scopes. For instance, in Figure 2, 2 instances of the `installment` function are created (i.e., `transferOut` and `transferIn`), each capturing the variable `_count` in its own parent scope created by 2 separate invocations of the `makeTransfer` function. These 2 scopes created by the `makeTransfer` function capture yet another variable `_balance` declared in a common parent scope corresponding to the *first invocation* of the `makeAccount` function (bound to the variable `alice`), but *not the second invocation* of the same function (bound to the variable `bob`). Thus, the reconstructed program should restore accordingly multiple instances of the `installment` function and their enclosing `Scope` objects, multiple instances of the `makeTransfer` function and their `Scopes`, and the relationship between the various `Scopes`.

Code Generation Algorithm. A simplified version of the code generation algorithm is shown in Algorithm 2. The `restore` function in line 1 is the user-facing function taking in a single argument `snapshot` and returning the generated code string. At a high-level, the program state captured in the snapshot have 2 parts: the scope tree representing the organization of the program's data, and the queue of timer events representing the program's control-flow state. Program restoration involves generating code to reconstruct the data state (line 2) and the control-flow state (line 3).

In a nutshell, the `generateCode` traverses the serialized representation of the scope tree and recursively generates code string, starting from the root (global) scope. Given a scope, the first line of the code generated is the code to initialize the `Scope` itself (line 7). In contrast to the code injected during the instrumentation, this restored `Scope` receives an extra `uid` argument to reassign the same instance ID it had when the snapshot was taken. This ensures that other objects dependent on the scope can correctly retrieve the same scope instances. Next, the functions (closures) defined in the scope are injected, again wrapped with the `Scope.addFunction` call (lines 8-10). Then the local variables of the scope are injected, each initialized with *the same value it had at the time of snapshot* (lines 12-19). For some variables during this step, it might happen that the scopes they depend on have not been reconstructed yet. An example would be a variable storing a closure function, which was originally created by one of the child scopes. Without generating the child scope first, the closure function cannot exist and therefore cannot be bound to the said variable. Our approach in TREECOPY addresses these situations by pushing such variables in a queue, to process them at a later step (i.e., after the generation of the child scopes - lines 20-22). After the first pass through the variables, `generateCode`

```
1  function restore(snapshot: JSON) : String
2      dataCode ← generateCode(snapshot.root)
3      timerCode ← generateTimers(snapshot.timers)
4      return wrapTemplate(dataCode + timerCode)
5  end
6  function generateCode(scope: JSON) : String
7      code ← "var " + scope.name + " = new Scope(" + scope.parent + ", '" + scope.uid + "');"
8      foreach func in scope.funcs do
9          code ← code + scope.name + ".addFunction(" + func.toString() + ");"
10     end
11     stage2 ← []
12     foreach variable in scope.vars do
13         if hasDependentScope(variable) then
14             stage2.push(variable)
15         else
16             code ← code + "var " + variable.name + " = " + variable.value + ";"
17             code ← code + scope.name + "." + variable.name + " = " + variable.name + ";"
18         end
19     end
20     foreach childScope in scope.children do
21         code ← code + generateCode(childScope)
22     end
23     foreach variable in stage2 do
24         code ← code + "var " + variable.name + " = " + variable.value + ";"
25         code ← code + scope.name + "." + variable.name + " = " + variable.name + ";"
26     end
27     return "(function (" + names(scope.params) + "){" + code + "})(" + values(scope.params) + ")"
28 end
29 function generateTimers(timers: Iterable) : String
       /* skipped for brevity                                                        */
30 end
```

Algorithm 2: Code Restoration Algorithm

function is called recursively on all child scopes. The last step is to restore the variables that were deferred in the first step (lines 23-26). Finally, before the code generated so far is returned, the entire code is *wrapped* as a function invocation, with the same function signature as the original function that created the scope and with the arguments it was invoked with (line 27). This step ensures that the closures created in the scopes are referencing the correct arguments. When the recursive `generateCode` call finally returns from its invocation on the root scope, the returned code string can be called to recreate the scope hierarchy.

We skip the details of `generateTimers` as they are quite trivial. For example, if a 1000ms `setInterval` timer has 750ms left, we first generate a `setTimeout` call with 750ms and then invoke the `setInterval` inside the function passed to `setTimeout`.

Once the code for the scope tree and the timers are available, they are concatenated and then *wrapped* with the ThingsMigrate bootstrapping code template in a similar way as in the instrumentation phase (line 4). As a result, the restored code contains the same set of injected ThingsMigrate objects and can be migrated again in the same fashion.

Code Restoration Example. Assume that a snapshot was taken after executing the code shown in Figure 2 for 3250 milliseconds. Figure 6 illustrates the restored code. Note that while this example has been derived from the output of a real invocation of the code restoration procedure of ThingsMigrate, some simplifications and adjustments were made for clarity. Also, the names of the various entities within this snippet (i.e., variables, functions, scopes), as well as their relationships, correspond to the state example shown in Section 4.5.

First we see `bob`'s scope being created by an invocation of the `makeAccount` function with the same arguments it was first called with (lines 4-10). The closed variable `_balance` is assigned the value 40, which it had at the time of snapshot after 3250 ms of execution. Following the scope generation, the global variable `bob` is assigned an object with its properties referencing the closure functions in `bob`'s scope that was just restored (lines 11-15). A different scope for `alice` is created, initializing the variable `_balance` with the value 60. In addition, there are 2 child scopes created, each containing the closure `installment` for the 2 different invocations of `alice.makeTransfer`. In the first `installment` scope (lines 23-28), the closed variable `_count` is initialized to 3 (`transferOut` was called 3 times) and in the second scope (lines 29-34) it is initialized to 2 (`transferIn` was called 2 times). The global variable `alice` is initialized in a similar manner to `bob` (lines 36-40). The closures bound to

```
1  var σ0 = new Scope();
2  function makeAccount(name, initial){ /* makeAccount function body */ };
3  σ0.addFunction(makeAccount);
4  (function makeAccount(name, initial){
5     var σ1 = new Scope(σ0, 'bob');
6     σ1.addFunction(function λ1(amount){ /* balance (λ1) function body */ });
7     σ1.addFunction(function λ2(account, amount, repeat){ /* makeTransfer (λ2) function body */ })
8     var _balance = 40;
9     σ1.vars._balance = _balance;
10 })('Bob');
11 var bob = {
12    name: 'Bob',
13    balance: σ0.getFunction('bob.λ1'),
14    makeTransfer: σ0.getFunction('bob.λ2')
15 };
16 σ0.vars.bob = bob;
17 (function makeAccount(name, initial){
18    var σ1 = new Scope(σ0, 'alice');
19    σ1.addFunction(function λ1(amount){ /* balance (λ1) function body */ });
20    σ1.addFunction(function λ2(account, amount, repeat){ /* makeTransfer (λ2) function body */ })
21    var _balance = 60;
22    σ1.vars._balance = _balance;
23    (function λ2(account, amount, repeat){
24       var σ3 = new Scope(σ1, 't1');
25       σ3.addFunction(function installment(){ /* installment function body */ });
26       var _count = 3;
27       σ3.vars._count = _count;
28    })(bob, 20, 4);
29    (function λ2(account, amount, repeat){
30       var σ3 = new Scope(σ1, 't2');
31       σ3.addFunction(function installment(){ /* installment function body */ });
32       var _count = 2;
33       σ3.vars._count = _count;
34    })(bob, -10, 6);
35 })('Alice', 100);
36 var alice = {
37    name: 'Alice',
38    balance: σ0.getFunction('alice.λ1'),
39    makeTransfer: σ0.getFunction('alice.λ2')
40 };
41 σ0.vars.alice = alice;
42 var transferOut = σ0.getFunction('alice/t1.installment');
43 σ0.vars.transferOut = transferOut;
44 var transferIn = σ0.getFunction('alice/t2.installment');
45 σ0.vars.transferIn = transferIn;
46 σ0.setTimeout(function(){
47    transferOut();
48    σ0.setInterval(transferOut, 1000);
49 }, 725);
50 σ0.setTimeout(function(){
51    transferIn();
52    σ0.setInterval(transferIn, 1500);
53 }, 1250);
```

FIGURE 6 JavaScript Code Example - instrumented with V1

`transferOut` and `transferIn` are retrieved from the reconstructed scope tree (lines 42, 44). Finally, the 2 timer states are restored to resume execution.

Multiple Migrations. ThingsMigrate supports transparent multiple migrations without introducing additional overhead (challenge 6). This is accomplished at the code restoration phase by maintaining a unique scope tree structure that is accessed by all the generated closures and scopes, and by re-injecting scope definitions (i.e., variables, parameters, nested functions, etc.) across the regenerated code, following an approach derived from Algorithm 1. Further, relevant Pub/Sub code is re-injected to support receiving *migrate* messages again. In other words, the output of the code restoration phase is an alternate code segment equivalent to the output of the code instrumentation phase, which can hence support further migrations.

5.1.4 | Limitations

Handling External Libraries. ThingsMigrate does not yet provide full support for imported libraries (i.e., the `require` statement). A simple solution would be to directly import the code in the main JavaScript module itself prior to instrumentation. This approach may be inefficient however, if there are multiple levels of nested library imports. Another solution would be for ThingsMigrate to provide a migration interface, and for module developers to implement the interface for either a more optimized migration of the nested libraries, or for supporting libraries exposing native I/O resources, such as file system access. Despite this limitation, we find that ThingsMigrate can support many third-party libraries as we show in Section 7.

Scope Explosion. If programs make use of several levels of nested closures, then the resulting snapshot and restored code can become quite large, due to the phenomenon of *scope explosion*, in which multiple scopes might have to be maintained. However, this problem is symptomatic of bad programming practices and is not specific to ThingsMigrate, as the JavaScript VM itself will have to retain a large amount of scope structures in-memory.

Redirecting I/O Operations. As mentioned in Section 4.2, ThingsMigrate assumes that all communications are done over the Pub/Sub interface. Further, in the current state, ThingsMigrate does not support file I/O operations, which is non-trivial, as reads and writes must occur where the corresponding files are located. For instance, assume there is a file on device A which is read by an application on the same device that gets migrated to device B. In order to guarantee consistent reads, we must guarantee (1) the availability of the file on B, or (2) to provide some redirection mechanism.

As JavaScript I/O operations are typically handled through streams, we plan on transparently redirecting streams over the Pub/Sub interface (solution 2 above), by wrapping the base JavaScript stream API (similar to wrapping timer-based or Pub/Sub-based APIs). A stream-level solution can support arbitrary stream-based I/O operations, such as files, network, and even HTTP requests. Upon device A receiving a migration request to migrate a given app to device B, the ThingsMigrate Runtime will generate a unique ID for each currently active stream, and will setup a transparent forwarding mechanism over a Pub/Sub bridge (i.e., by creating a topic corresponding to that ID that both devices A and B will subscribe to). Then, upon a read operation being requested by the app on device B, for a given stream, the request will be transparently forwarded by the Runtime to device A, which will perform the read and send back the results to the Runtime on B, who will deliver them to the stream at the application layer. Likewise, any write operation will simply be forwarded from the Runtime on B to the Runtime on A.

Nested Timers. Another limitation is the handling of some deeply nested timer-related calls (i.e., `setTimeout`, `setImmediate`). Should a *snapshot* command be received while a *timer* is in a *pending* state – i.e., before the callback function is invoked – then the timer gets cleared, the remaining time and the reference to the callback function are serialized, and migration happens normally. However, should the snapshot command be received *after* the callback function is invoked, then a race condition occurs between any asynchronous calls made inside the body of the callback and the snapshot function. Race conditions are sometimes problematic in JavaScript, as the ordering of events is unpredictable[52,53]. For instance, should the JavaScript VM event loop process the snapshot function before the asynchronous calls, then the resulting snapshot will not contain the scopes created by the asynchronous calls, producing an incorrect snapshot. Handling nested timers would require that the snapshot function be delayed until all callbacks have been resolved, which is a non-trivial problem. To address this, we can inject at the instrumentation phase, specific code into the function scope that will signal the function's completion, which would allow us to detect the resolution of nested asynchronous calls.

5.2 | Technique 2: "XPLICTGC"

We successfully validated our approach with TREECOPY; however, the instrumented (migration-enabled) programs were consuming more than double the memory due to the duplication of state, and were significantly slower because of the additional work done in tracking the changes in program state. We came up with a new technique (XPLICTGC) to address the following optimization goals: (1) to reduce the memory footprint, (2) improve run-time performance, (3) reduce the snapshot and code restoration time to minimize the overall migration latency, and (4) shorten the instrumentation time.

We focus our discussion of XPLICTGC on the code instrumentation phase, as the snapshot and restoration steps remain mostly the same. In TREECOPY, duplicating the logical program state incurs a significant overhead both in terms of memory usage and run-time speed. The instrumented program consumes at least twice the amount of memory used by the original program due to keeping an explicit copy of each variable. The extra code injected for updating the explicit state consumes additional CPU cycles doing bookkeeping work; the single-threaded nature of JavaScript makes this overhead very apparent.

Keeping a single copy of the state. To address both of the above issues, the first idea was to find a way to work with a single copy only. Since we need to be able to access every object (including those inside closures) for capturing the program

```
var σ0 = new Scope();
function makeAccount(name, initial){
   var σ1 = new Scope(σ0);
   σ1.vars._balance = initial || 0;
   return {
      name: name,
      balance: σ1.addFunction(function λ1(amount){
         var σ2 = new Scope(σ1);
         if (typeof amount === 'number'){
            σ1.vars._balance += amount;
         }
         return σ1.vars._balance;
      }),
      makeTransfer: σ1.addFunction(function λ2(account, amount, repeat){
         var σ3 = new Scope(σ1);
         σ3.vars._count = 0;
         return σ3.addFunction(function installment(){
            var σ4 = new Scope(σ3);
            if (σ3.vars._count < repeat){
               σ1.vars._balance -= amount;
               account.balance(amount);
               σ3.vars._count ++;
            }
            console.log(name+' $' + σ1.vars._balance + ', ' + account.name + ' $' + account.balance());
         })
      })
   }
}
σ0.addFunction(makeAccount);
σ0.vars.alice = σ0.vars.makeAccount('Alice', 100);
σ0.vars.bob = σ0.vars.makeAccount('Bob');
σ0.vars.transferOut = σ0.vars.alice.makeTransfer(σ0.vars.bob, 20, 4);
σ0.vars.transferIn = σ0.vars.alice.makeTransfer(σ0.vars.bob, -10, 6);
σ0.setInterval(σ0.vars.transferOut, 1000);
σ0.setInterval(σ0.vars.transferIn, 1500);
```

FIGURE 7 JavaScript Code Example - instrumented with V2

state, we decided to keep the explicit copy, and get rid of the original scope tree. This entails the following modifications to the instrumentation algorithm introduced in TreeCopy:

1. Remove all injected AssgnExpr that follow an *explicit-change* statement (refer to Table 2).
2. Replace all VarDecl with an AssgnExpr on the corresponding scope property. For example, `var foo = "bar"` declared in scope `scope_0` becomes `scope_0.vars.foo = "bar"`.
3. In all statements in the scope, replace Idnt of the original variable with the MembExpr of the corresponding scope property. For example, all references to `foo` are replaced with `scope_0.vars.foo`.

Figure 7 shows the instrumented example program using XplictGC. The local variable `_balance` is an explicit property in the `vars` object of the corresponding `Scope` ($\sigma 1$). Similarly, the variable `_count` is bound to σ`3.vars`. Additionally, all the references to `_balance` and `_count` are replaced with the respective MembExpr. As a result, the instrumented program resembles the original program. In a nutshell, we essentially remove all the free variables from the program and turn them into properties of the enclosing `Scope` object.

Preserving the lexical semantics. Because we replace all the Idnt with the corresponding MembExpr, more work needs to be done during the code instrumentation phase. Apart from performing an increased number of AST node modifications, the *Instrumentor* must also preserve the lexical semantics of JavaScript. The first issue is that VarDecl and FuncDecl are *hoisted*, while AssgnExpr are not. Because *hoisted* statements are processed before everything else within the execution context even if they are placed at the end, we cannot naively change a VarDecl to an AssgnExpr without inspecting the order of statements first. Next, the lexical scope of each Idnt needs to be tracked during the instrumentation, since we no longer rely on the native VM for resolving the lexical bindings. For example, assuming there is an extra global variable `_balance` in the running example, the *Instrumentor* has to track whether a given identifier `_balance` is referring to the variable in the global scope or to the local variable inside `makeAccount`'s scope.

Handling Garbage Collection (GC). When we implemented the above approach, we found there was a noticeable degradation in terms of memory usage and execution speed. Since we have attached all the state variables explicitly as properties of scope objects, all the variables created in the program were reachable from the global scope. This meant that no variables were automatically garbage collected, even if they were no longer used by the program. To make objects available for garbage collection, XPLICTGC had to dereference variables that are no longer needed by the program. Determining which objects can be safely GC'ed is a difficult problem to solve during run-time, so we instead perform a Mark-and-Sweep periodically to collect the active scope objects. The scope objects that are not referenced by any of the objects currently in scope are dereferenced and made available for the native GC. To clarify, this Mark-and-Sweep performed by XPLICTGC is not an actual GC, but a "pre-GC" to make objects available for the native JavaScript VM's GC.

Lesson learned. Roughly speaking, what we end up building in this approach is another JavaScript VM written in JavaScript. Instead of relying on the native VM (i.e., Node.js) to handle the lexical binding of variables, the *Instrumentor* needs to resolve the binding. In the instrumented program, every variable is accessed via properties of scope objects, which is slower than accessing via direct references. When there are many transient objects, the Mark-and-Sweep is an expensive operation to perform periodically. In XPLICTGC, the performance gain due to lower memory usage is minimal, and the overall performance degrades.

5.3 | Technique 3: "LAZYSNAP"

Although our approach in XPLICTGC was not successful, the experience of building the Mark-and-Sweep mechanism revealed an important insight: *Mark-and-Sweep essentially produces a snapshot of the active program state*. Another important insight – recurrent in various domains of optimizations – is to track the state lazily. The end-goal in our migration problem is simply to be able to capture the program state on-demand; actively mirroring the state is not necessary. Combining the two insights, we identified that a more efficient way to achieve this goal would be to *invoke Mark-and-Sweep lazily upon a snapshot command* and trace the active scope tree to capture the state (i.e., lazily tracking the state).

Before we discuss further, we briefly describe the Mark-and-sweep GC mechanism used by Node.js V8 engine. This GC algorithm is the *de facto* GC used in most modern JS engines. The high-level intuition underlying the Mark-and-Sweep algorithm is to traverse the "links" between the objects in the heap and carve out a graph of reachable objects, starting from the objects directly accessible from the root scope. The objects that are not included in the graph are unreachable by the program and thus can be safely garbage collected. Figure 3 illustrates the algorithm[54].

```
function markAndSweep(items: Set, marked: Set) : Set
    foreach item in items do
        if $item \in marked$ then
            continue
        else
            $marked \leftarrow marked \cup \{item\}$
            if $item \notin$ LITERAL then
                markAndSweep(getScope(item), marked)
                markAndSweep(getProperties(item), marked)
            end
        end
    end
    return marked
end
$reachable \leftarrow$ markAndSweep(rootScope, $\emptyset$)
$unreachable \leftarrow$ heap $\oplus$ reachable
foreach object in unreachable do
    delete object
end
```

Algorithm 3: Mark-and-Sweep Algorithm

Lazily capturing the original scope tree. As per our assumptions, we do not have access to the JavaScript VM and its GC, so we cannot leverage the native Mark-and-Sweep mechanism. Thus, the challenge of accessing enclosed variables still remains, if we build our own Mark-and-Sweep. To expose the closed variables *lazily*, we inject *a callback function in the variable's lexical scope, which we can call later to access the variables*. To reiterate, instead of injecting code that actively copies the values of

```
var σ0 = new Scope();
σ0.extract = function(){
   return {
      makeAccount: makeAccount,
      alice: alice,
      bob: bob,
      transfer: transfer
   }
};
function makeAccount(name, initial){
   var σ1 = new Scope(σ0);
   σ1.extract = function(){
      return {
         _balance: _balance
      }
   };
   var _balance = initial || 0;
   return {
      name: name,
      balance: σ1.addFunction(function λ1(amount){
         if (typeof amount === 'number'){
            _balance += amount;
         }
         return _balance;
      }),
      makeTransfer: σ1.addFunction(function λ2(account, amount, repeat){
         var σ3 = new Scope(σ1);
         σ3.extract = function(){
            return {
               _count: _count
            }
         };
         var _count = 0;
         return σ3.addFunction(function installment(){
            if (_count < repeat){
               _balance -= amount;
               account.balance(amount);
               _count ++;
            }
            console.log(name+' $' + _balance + ', ' + account.name + ' $' + account.balance());
         })
      })
   }
}
var alice = makeAccount(100);
var bob = makeAccount();
var transferOut = alice.makeTransfer(bob, 20, 4);
var transferIn = alice.makeTransfer(bob, -10, 6);
σ0.setInterval(transferOut, 1000);
σ0.setInterval(transferIn, 1500);
```

FIGURE 8 JavaScript Code Example - instrumented with V3

the variables, we inject a callback function, which returns the copy of the variables when invoked. Figure 8 shows the example code transformed using LazySnap.

In this approach, the callback function resides in the same function body as the variables it is capturing. The lexical binding of all the variables remain unchanged from the original code, and the callback function inherits the same lexical scope as the variables. Unless invoked, the injected callback does not interfere with the normal execution of the program and thus introduces minimal run-time overhead. At any point during the execution, a `Scope`'s `extract` callback can be invoked to retrieve its enclosed variables. With this modification, we have avoided actively tracking the state, and are able to capture all the variables lazily on demand.

Aligning the lexical scope of injected objects for natural GC. At this point, we have effectively made the migration technique work lazily. However, the injected objects created during run-time still need to be explicitly dereferenced. The best strategy would be to avoid having to dereference objects altogether, and let the native GC collect unused objects. To achieve this, the next step is to organize the injected code in such a way that the life-cycle of the injected objects is aligned with the actual life-cycle of the enclosing function. That is, if a closure goes out of scope naturally, its corresponding `Scope` object should too.

Concretely, we update the definition of the Scope object so that there is only a one-way reference between a parent and a child Scope. Unlike TreeCopyand XplictGC, a parent Scope in LazySnap does not keep references to its children, whereas the child Scopes can reach the parent. This aligns with the operational semantics of JavaScript– objects in the child scope can refer to variables declared in the parent scope, while the parent cannot access the closure variables in the child scope. In a similar fashion, the child Scope has access to the parent Scope, but the parent Scope has no direct link to the child Scopes. After this modification, if a child Scope is not reachable from the root scope, they are naturally garbage collected; ThingsMigrate does not need to manage the life-cycle of objects.

On the other hand, without a direct reference from the root Scope, certain descendant Scopes need to be reachable when capturing a snapshot; we make them reachable via *objects currently in scope*. A scope is needed only if it is associated with an object in the reachable program context. So on every non-literal object (e.g., objects, functions), we attach a "private" property and bind the associated Scope object. Then at the time of snapshot, we can traverse the Scope tree starting from the root, using our adaptation of Mark-and-Sweep as shown in Algorithm 4.

```
function capture(scope: Scope) : dict
    if scope.visited then
        return
    else
        scope.visited ← True
        scope.image ← { refs, children }
        refs ← scope.extract()
        foreach name, object in refs do
            scope.image.refs[name] ← serialize(object)
            captureObject(object, scope)
        end
        if scope.parent then
            capture(scope.parent)
            scope.parent.image.children[scope.id] ← scope.image
        end
        scope.visited ← False
        return scope.image
    end
end
function captureObject(object: Object, scope: Scope) : void
    if isNotLiteral(object) then
        if isFunction(object) and object._parentScope != scope then
            capture(object._parentScope)
            foreach property in object.prototype do
                captureObject(property, scope)
            end
        else if isObject(object) and isDefined(object._scope) then
            capture(object._scope)
        end
        foreach property in properties(object) do
            captureObject(property, scope)
        end
    end
end
snapshot ← capture(rootScope)
```

Algorithm 4: LazySnap Snapshot Algorithm

Finally, we also skip instrumenting a function if we can statically determine that it only creates transient state (e.g., pure functions). That is, if a function does not create any objects and does not return an object other than literals, we can safely assume that it does not create any closures and therefore need not be tracked. Inside such a function, we do not even inject the Scope object, and leave the entire function intact.

5.4 | Implementation

ThingsMigrate is implemented in the form of a JavaScript library that can be included by the application. Its implementation is built over the ThingsJS system, and is part of ThingsJS[81]. It provides APIs that can be invoked to perform code instrumentation, snapshotting and code restoration.

From a higher-level perspective, our implementation also provides an execution environment that replicates the architecture shown in Figure 1. More specifically, it provides a runtime environment that can be run on IoT devices supporting an appropriate VM (e.g., Node.js on Raspberry PIs Models 3 and 0), as well as a *Manager* component, which is used to transparently instrument JavaScript programs, launch them on specific IoT nodes (decided by a scheduler), monitor them, and trigger a serialization/migration. Internally, our implementation uses the popular `esprima` library[55] to parse JavaScript code into an AST, and the `escodegen`[56] to convert back an AST into JavaScript code.

6 | EXPERIMENTAL VALIDATION

We perform 4 sets of experiments to evaluate ThingsMigrate. Experiment 1 (Section 6.2) measures the performance of our code instrumentation algorithm against a set of benchmarks. Experiment 2 (Section 6.3) measures the run-time performance overhead of ThingsMigrate in terms of execution time and memory usage. Experiment 3 (Section 6.4) measures the migration latency comprising the snapshotting and code generation time. Finally, Experiment 4 (Section 6.5) evaluates the multi-hop migration capabilities of ThingsMigrate by migrating a benchmark application several times across different devices.

6.1 | Experimental Setup

ThingsMigrate provides JavaScript migration between IoT devices, and between devices and the *cloud*. To emulate different scenarios, we ran our experiments on two IoT platforms, namely a Raspberry Pi model 3B (quad-core 1.2 Ghz ARM7, 1 GB memory), and a Raspberry Pi model 0W (single-core 1 Ghz ARM6, 512 MB memory), both running the Raspbian Jessie operating system (a Debian Linux variant). We also included a *cloud* server (Xeon E3-1220 v3, quad-core 3.10Ghz, 32 GB memory). All nodes were running the Node.js VM version 8, which is ES5 compliant. While we did not test other VMs due to stability issues or due to incompliance with the ES5 standard, each of the Node.js VMs we used were compiled for different architectures (armv7, armv6, and x86-64 respectively).

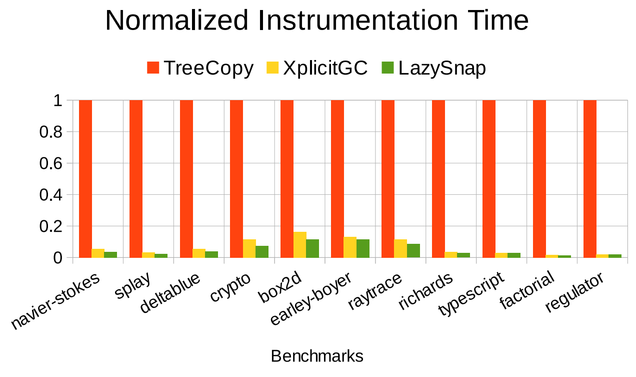
Despite an extensive search, we did not find publicly-available sets of IoT-specific JavaScript benchmarks to evaluate our system. Prior work ([14]) has built their own IoT-specific JavaScript benchmarks[2]. We followed a similar approach and built two IoT-specific benchmarks: (1) a *factorial* application, which computes the factorial of a very large number and uses closures to store the computed digits (i.e., in a very large expanding array), and (2), a *regulator* application, which models an IoT *edge* component which receives temperature measurement data from different sensors[3] over a Pub/Sub interface, keeps the previous n values for m sensors, and periodically computes an optimal power adjustment to be sent to an actuator. `factorial` models a CPU and memory-intensive application of a finite duration (experiment 2), while `regulator` models a less intensive (i.e., low CPU and memory usage) application that runs for a long time. Both applications are stateful, and need to preserve state across migrations. Note that the memory usage of the `regulator` is similar to the memory usage of typical IoT-specific benchmarks[14].

In addition, for experiments 1 and 2, we also used some benchmarks from the Chromium Octane[57] suite, which were originally designed to stress-test the performance of the V8 JavaScript engine in the Chrome web browser. They run *synchronously* and hence are not representative of IoT applications, but we nevertheless use them to assess the universality of our framework, and for performance testing of ThingsMigrate under intense workload.

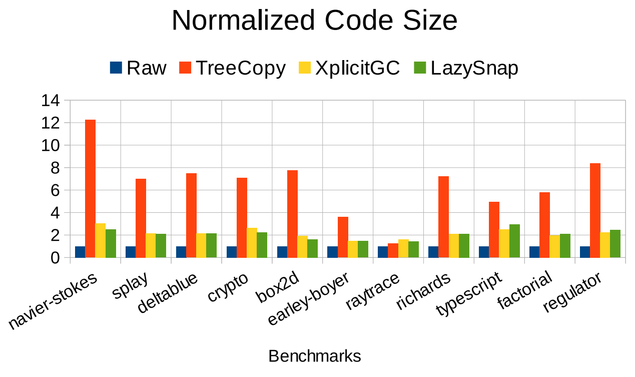
6.2 | Experiment 1: Code Instrumentation

In this experiment, we consider all the benchmark programs from the Chromium Octane suite that do not depend on a web browser (i.e., accessing the DOM or any other in-browser object), as ThingsMigrate migrates IoT applications (i.e., *server-side*)

[1]http://www.github.com/DependableSystemsLab/ThingsJS
[2]The source code is not publicly available, and hence we cannot use them.
[3]We fed the application with pre-determined values, as the computed result itself is not part of the experiment.

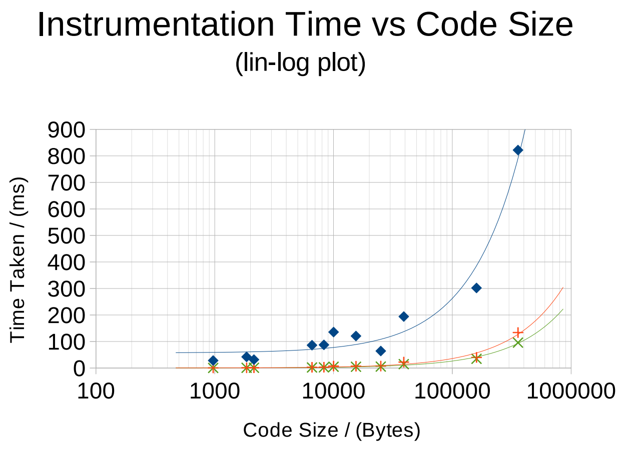


(a) Time Taken (Lower is better)

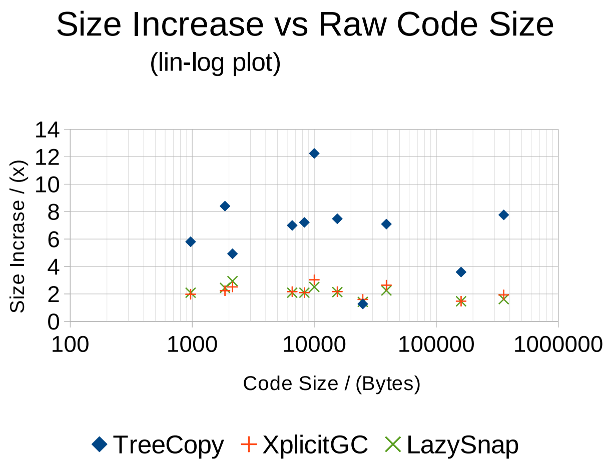


(b) Code Size Increase (Lower is better)

FIGURE 9 Code Instrumentation Results (with a confidence interval of 95%)



(a) Time Taken (Lower is better)



(b) Code Size Increase (Lower is better)

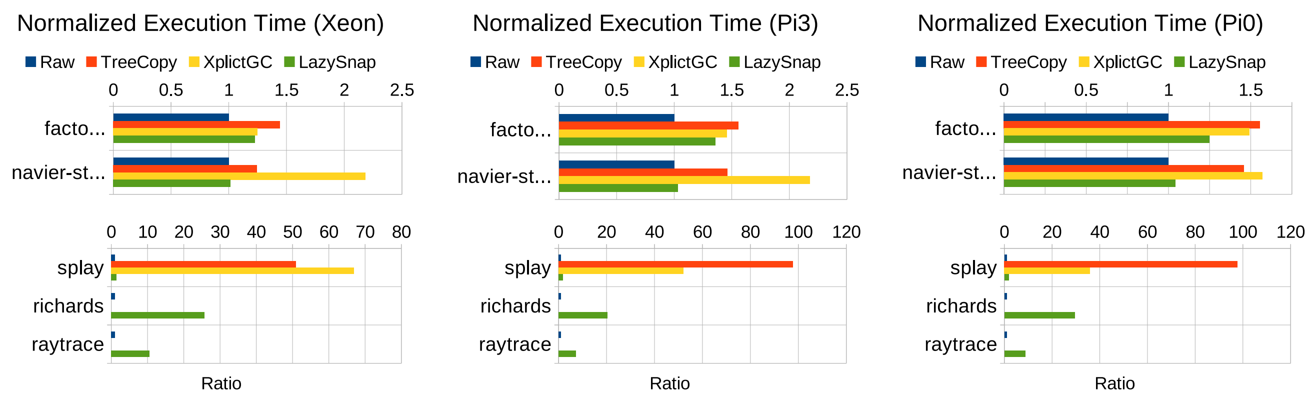
FIGURE 10 Code Instrumentation Results versus Code Size

rather than in-browser applications. We measure the time it takes to instrument the code for these benchmarks[4], as well as for our `factorial` and `regulator` applications. In addition, we compare the size of the instrumented code against the size of the uninstrumented (raw) code. The set of benchmarks and the set of results of instrumentation using all 3 approaches (TREECOPY, XPLICTGC, LAZYSNAP) are shown in Table 3 and Table 4.

Figure 9a shows the time taken to instrument each benchmark, normalized to the time taken by TREECOPY. The main takeaway of this plot is the significant improvement in the performance of the instrumentation algorithm. As described in Sections 5.2 and 5.3, we inject less code in XPLICTGC and LAZYSNAP compared to TREECOPY where we inject a follow-up expression for every ASSGNEXPR, which makes it much faster.

Figure 10a shows a lin-log (i.e., log linear) plot of intrumentation time against code size for all 3 approaches. The instrumentation time for TREECOPY is noticeably larger than those for XPLICTGC and LAZYSNAP. Approaches XPLICTGC and LAZYSNAP yield similar results, with LAZYSNAP performing marginally better. We also confirm that the instrumentation time grows linearly with the code size – i.e., they fit the curve of the form $f(x) = ax + b$ (the plot for TREECOPY is approximated by the curve $f(x) = 2.055 \times 10^{-3}x + 56.998$, XPLICTGC by the curve $f(x) = 0.355 \times 10^{-3}x + 0.327$, and LAZYSNAP by the curve $f(x) = 0.259 \times 10^{-3} + 0.404$). Using the functions, we can estimate that instrumenting a program of 1MB would take

[4]Measurements were taken on our *cloud* server.



(a) *cloud* Server (Xeon) **(b)** Raspberry Pi 3 **(c)** Raspberry Pi 0

FIGURE 11 Normalized Execution Time (Lower is better). Margins of errors were below 1.5% for most of our results, and up to 6% for some of our results on the Pi 0, for a confidence interval of 95%.

about 263ms using TREECOPY, and about 26ms using LAZYSNAP, which is an order of magnitude faster. Note that even in our slowest approach (TREECOPY), the instrumentation is performed quite quickly (in under 1 second) for the benchmark applications. Further, code instrumentation is a one-time process for any given program and can always be performed on a machine with higher compute capacity, or can be directly integrated as part of the build process, similar to code minification.

The size of the instrumented code relative to the original is shown in figure 9b. In TREECOPY, the median code size increase is around 700%, while in XPLICTGC and LAZYSNAP the median code size is around 200%. In Figure 10b we provide a lin-log plot of the size increase as a function of the original code size.

6.3 | Experiment 2: Run-time Performance Overhead

In this experiment, we analyze the performance impact of ThingsMigrate over a set of highly resource-intensive benchmarks. The goal of this experiment is to model the execution of a resource-intensive *task* of a finite duration (i.e, eventually returns a result) that would be executed over different IoT devices and the *cloud* server. For evaluating the 3 techniques, we selected benchmarks `navier-stokes` and `splay` from the Octane suite, as they respectively model extreme conditions in terms of CPU usage and memory utilization. Further, we were successful in running these benchmarks on all test devices and across all 3 versions of ThingsMigrate, unlike most other benchmarks in the suite (even without our instrumentation, most of the benchmarks in the Octane suite were unable to run on the Rapsberry Pi 0 due to its limited capabilities). We also used our `factorial` application. For the LAZYSNAP technique, we were able to additionally run `richards` and `raytrace` from the Octane suite, as LAZYSNAP addresses the limitations of the previous versions (e.g., multiple levels of prototypal inheritance).

For each benchmark, we measure and compare the time to complete its execution. On our 3 target devices (Raspberry Pi 3, Pi 0 and our *cloud* server), we run (A) the non-instrumented (raw) code, (B) the instrumented code, and (C) the restored code after migration. As the restored code (C) only runs the second half of the program (the snapshot is taken at the mid-point of the execution), we also only consider the second half for (A) and (B) for fairness. In addition, we measure the average memory usage of each of the runs. We perform this experiment for all 3 approaches, and compare the run-time and memory overhead incurred by each of the approaches.

Execution Time. Results for the execution time of selected benchmarks running on the *cloud* (Xeon), Pi3, and Pi0 are shown in Figure 11a, 11b, and 11c respectively. The execution times of the raw version of the benchmarks, represented by the blue bar, are normalized to 1. The execution time of the instrumented benchmarks using approaches TREECOPY, XPLICTGC, and LAZYSNAP are shown by the red, yellow, and green bars respectively.

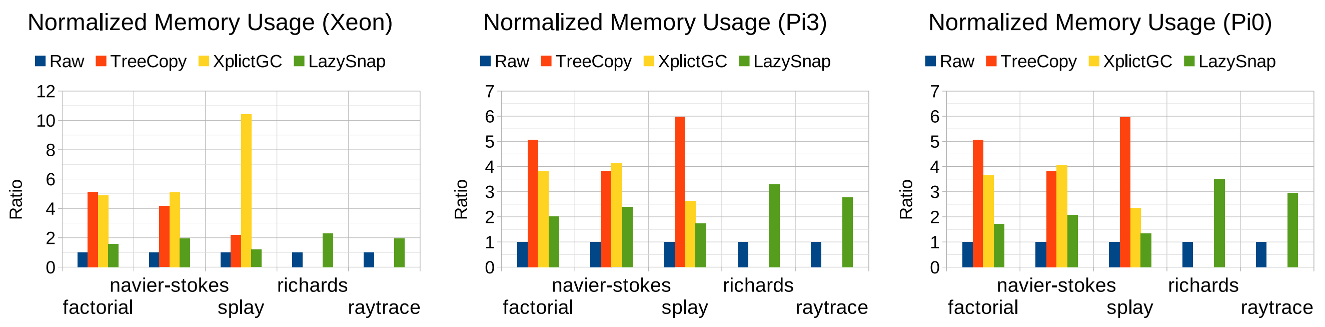
The results for the `factorial` program clearly illustrates the improvement over the 3 versions of ThingsMigrate. When using TREECOPY across all 3 devices, `factorial` slowed down by a factor of about 52% (averaged across 3 devices). XPLICTGC observes only a marginal improvement. However, in LAZYSNAP the average overhead is reduced to about 27%, where the overhead on the *cloud* is 23%.

Benchmark	Code Size (kb)	Time spent on GC	AST Node Distribution			% of State-changing
			NO-CHANGE	EXPLICIT-CHANGE	IMPLICIT-CHANGE	EXPRESSIONS
navier-stokes	9.985	0.0%	1428	304	81	21.2%
splay	6.573	13.2%	785	142	82	22.2%
deltablue	1.5452		1916	325	218	22.1%
crypto	39.028	0.1%	6558	1002	592	19.6%
box2d	357.169	1.3%	57288	7049	2534	14.3%
earley-boyer	159.794	17.0%	11201	1679	2901	29.0%
raytrace	24.998	5.6%	3009	348	232	16.2%
richards	8.302	0.5%	1122	212	99	21.7%
typescript	2.138		297	71	39	27.0%
factorial	0.952	0.0%	122	25	16	25.2%
regulator	1.855	1.7%	155	46	27	32.0%

TABLE 3 Benchmarks

Benchmark	TREECOPY		XPLICTGC			LAZYSNAP		
	Size Increase	Time Taken (ms)	Size Increase	Time Taken (ms)	Time Taken (as % of TREECOPY)	Size Increase	Time Taken (ms)	Time Taken (as % of TREECOPY)
navier-stokes	1220%	135.67 ± 4.5	300%	7.4 ± 0.7	5.4%	250%	4.8 ± 0.6	3.5%
splay	700%	86.18 ± 2.7	220%	2.8 ± 0.1	3.2%	210%	2.0 ± 0.1	2.4%
deltablue	7480%	120.65 ± 0.6	220%	6.8 ± 0.3	5.6%	210%	4.7 ± 0.1	3.9%
crypto	710%	194.05 ± 1.6	260%	22.4 ± 0.5	11.6%	230%	14.6 ± 0.2	7.5%
box2d	780%x	821.95 ± 3.0	190%	133.8 ± 1.7	16.3%	160%	95.8 ± 1.3	11.7%
earley-boyer	360%	301.9 ± 0.8	150%	40.0 ± 0.7	13.2%	150%	34.8 ± 0.6	11.5%
raytrace	130%	64.2 ± 0.5	160%	7.4 ± 0.2	11.5%	140%	5.5 ± 0.1	8.5%
richards	720%	87.4 ± 0.5	210%	3.1 ± 0.1	3.5%	210%	2.5 ± 0.1	2.8%
typescript	490%	31.75 ± 0.2	250%	1.0 ± 0.1	3.1%	290%	0.9 ± 0.1	3.0%
factorial	580%	28.21 ± 0.2	200%	0.5 ± 0.1	1.6%	210%	0.4 ± 0.1	1.5%
regulator	840%	42.1 ± 0.4	220%	0.9 ± 0.1	2.1%	240%	0.8 ± 0.1	2.0%
Average	1270%		220%		7.0%	210%		5.3%

TABLE 4 Code Instrumentation Results (with a confidence interval of 95%)



(a) *cloud* Server (Xeon) **(b)** Raspberry Pi 3 **(c)** Raspberry Pi 0

FIGURE 12 Memory Usage (mb) (Lower is better). Margins of errors are not shown, as the results show the averaged memory usage for all runs, averaged over the duration of the experiment.

Running `navier-stokes` exacerbates the flaws of XPLICTGC, as it performs worse than TREECOPY across all 3 platforms. The `navier-stokes` benchmark mostly calls functions that perform arithmetic operations over several variables and over elements in a large array. Running a JavaScript profiler shows 90.3% of CPU time is spent on `project(u, v, p, div)`, `vel_step(u, v, u0, v0, dt)`, `advect(b, d, d0, u, v, dt)` functions alone, which are all such functions. In XPLICTGC, all the variables are converted to explicit properties of `Scope` objects – i.e., IDNT are converted to MEMBEXPRS – and hence it takes longer to access and update each of the state variables. Furthermore, XPLICTGC's ad-hoc GC mechanism needs to work diligently to explicitly dereference variables that have gone out-of scope. The execution time overhead of `navier-stokes` is brought down to about 3% in LAZYSNAP, as the improved instrumentation algorithm is able to identify and skip over the functions that do not need to be captured. As a result, only a few functions that generate closure state are instrumented, with the rest of the program being untouched.

We plot the `splay` benchmark separately, because it incurred significantly larger overhead in TREECOPY and XPLICTGC. The `splay` benchmark creates a *splay tree* spanning thousands of nodes and rapidly transforms the tree's structure. Unlike `navier-stokes`, it instantiates higher-level objects like `SplayTree.Node` that contain closure states. When instrumented, each of the nodes in the *splay tree* instantiates a ThingsMigrate `Scope` object, adding to the run-time overhead. TREECOPY and XPLICTGC experience a much larger average overhead of 8220% and 3870% respectively, since both approaches involve some form of GC duty (i.e., explicitly dereferencing unused objects via `delete`). In LAZYSNAP, the overhead is brought down to 67%. The observed difference between the approaches highlight the importance of leveraging the native GC.

In addition to the 3 benchmarks, we were able to run `richards` and `raytrace` in LAZYSNAP. In contrast to the other benchmarks, both `richards` and `raytrace` create higher-level objects using prototypal inheritance and the `new` expression (as opposed to object literals). TREECOPY and XPLICTGC were not able to correctly restore the program on a target device, due to their limited support for prototype inheritance. Thus, we report the results for the 2 additional benchmarks, but exclude them from the overall average, as we cannot compare their results across the different versions of ThingsMigrate.

Despite the large slowdown in TREECOPY and XPLICTGC, all 3 approaches were able to correctly migrate the benchmarks. We note however that these benchmarks are *synchronous* programs that were specifically designed to stress-test browsers on desktop computers, and do not represent typical *asynchronous* and *event-driven* applications that run on IoT devices.

We also observe that the performance of the instrumented code (B) and the restored code (C) is the same across all benchmarks – i.e., time taken for restored program is within the confidence interval of the time taken for instrumented code and vice versa. As the restored code is *semantically equivalent* to the original code, we obtain the same performance measurements as the instrumented (*pre-migration*) code. These results indicate that the instrumentation overhead does not accumulate and degrade the performance (i.e., execution time) over subsequent migrations (Section 6.4).

Unfortunately, we cannot directly compare our results with prior work in terms of execution time overhead, as Lo et. al.[21] measured such overheads for web applications on desktop computers, which do not exhibit the same workload characteristics as our benchmarks, and Kwon et. al.[22] does not report on the execution time overheads at all.

Memory Usage. Figures 12a, 12b, and 12c present the memory usage of the benchmarks across 3 different platforms. Similar to the results in Figure 11, the memory usage of the raw benchmarks (shown in blue) are normalized to 1; the red, yellow, and green bars represent the different versions of ThingsMigrate. For each benchmark, we uniform-randomly sampled the memory usage throughout the execution of the benchmark, and report the average memory usage. We apply uniform random sampling and not periodic sampling to avoid the sampling interval coinciding with the garbage collection interval, which may render biased results. Overall, our results reveal that the instrumented programs use considerably more memory than the original program, ranging from 18% in LazySnap to 940% in XplictGC. LazySnap incurs the smallest memory overhead for all benchmarks and across all platforms.

Since ThingsMigrate is a pure application-level solution that does not involve augmenting the JavaScript VM (unlike Kwon et. al.[22]), some memory overhead is unavoidable. For example, every injected `Scope` object takes up additional space in the heap. For each benchmark, the observed memory overhead exhibits a different trend, which we attribute to the varying logical structure of the benchmarks. For `factorial`, the memory overhead is reduced over each version of ThingsMigrate, with LazySnap incurring the least memory overhead of 77%. Interestingly, the average memory overhead for `navier-stokes` is higher by 48% in XplictGC than in TreeCopy. In `factorial`, there are only a few `Scope` objects while the captured object is large (i.e., a large array of numbers). In this program, the size of the program state dominates the memory usage. However in `navier-stokes`, the program state consists of a smaller array and a few literal values, while there are a lot of `Scope` objects created by transient function calls. In such programs, the creation of `Scope` objects dominates the memory usage. This is why we see higher memory usage in XplictGC even without keeping a replica of the program state. LazySnap does not incur memory overhead for tracking the program state, since it captures the program state lazily. Rather, the memory overhead in LazySnap comes from the injected `Scope` objects and the associated `extract` callback functions. The results for `splay` also illustrates the improvement over the 3 approaches, with the lowest overhead of 42% in LazySnap. Surprisingly, the overhead introduced by XplictGC is larger than TreeCopy on the *cloud* machine, but not on the other 2 devices. A potential explanation for this is the interaction between the garbage collection cycle and the memory sampling code. As we measure the resource usage from within the application, the sampling function is also pushed into the JavaScript VM's event queue. Hence, we know that the sampling function can only be invoked when the synchronous part of the benchmark has finished. During the synchronous part of `splay`, thousands of `Scope` objects are created, consuming a lot of memory. On the *cloud* device, the next synchronous part of `splay` can be queued right away, as there is still gigabytes of memory left. In contrast, on the Raspberry Pi devices, the VM invokes garbage collection more aggressively, to clean up the unused `Scope` objects and make room for the next function invocation. Therefore, there is a higher chance on the Raspberry Pi devices that the memory sampling function is invoked after the garbage collector has freed up memory. The memory logs support this explanation, in which the memory consumption for Raspberry Pis drops at regular interval, while the pattern observed on the *cloud* is more arbitrary.

In addition to the `Scope` objects injected throughout the user code, there are other ThingsMigrate objects initialized during the *bootstrapping* step (Section 5.1.1) that perform logistical work unrelated to the application logic such as maintaining an MQTT Pub/Sub connection. Thus, we further break down the overall memory usage into 2 parts: ① memory used for tracking the program state, and ② memory used by *helper objects* providing the Pub/Sub interface for triggering the migration. Part ① is the unavoidable overhead in any implementation of ThingsMigrate; `Scope` objects are injected regardless of how the migration is triggered and how the snapshot is transported. Part ② is the variable overhead and is mainly due to our choice of the MQTT Pub/Sub infrastructure for triggering the migration process. We therefore subtract the latter from the total memory overhead. We measured the overhead of part ② by executing an "empty" benchmark that performs no computation, and found it to be about 19MB for 64-bit devices and 9.5MB for 32-bit devices. We note that the absolute memory usage for both Raspberry Pi devices are half of that of the *cloud* server due to the *bitness* of the devices; i.e., our *cloud* server has a 64 bit processor, while the other Pi devices have 32 bit processors. Further, GC is triggered more aggressively on the Pi devices, as they are more memory-constrained.

6.4 | Experiment 3: Migration Overhead

In this experiment, we report the migration overhead consisting of the time taken to capture and serialize the state, and the time taken to generate restoration code. We ran each benchmark until it made 50% progress, took a snapshot, and then generated restoration code from the snapshot. As we noted previously, as the benchmarks were *synchronous* programs (i.e., do not yield control to the migration framework), we manually injected code to trigger the migration from within the benchmark; this also has the effect of migration being triggered deterministically at the desired point in the execution. Each benchmark was captured and restored multiple times on each platform, and the times averaged over the multiple runs are displayed in Figure 13.

Similar to the results of Experiment 6.2, we observe that XplictGC and LazySnap both perform significantly better than TreeCopy, mostly due to the improved implementation of the *Instrumentor* and *Runtime*. An interesting observation regarding

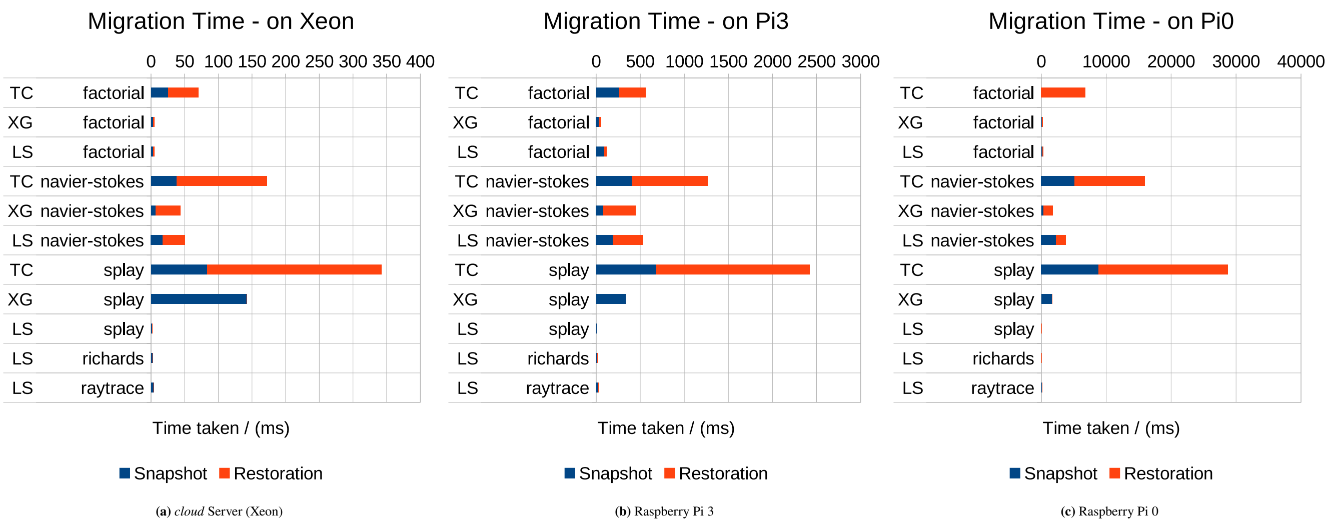


FIGURE 13 Snapshot / Restoration Time (in seconds). Margins of error were between 0.5% and 5% for all results, for a confidence interval of 95%. (TC = TREECOPY, XG = XPLICTGC, LS = LAZYSNAP)

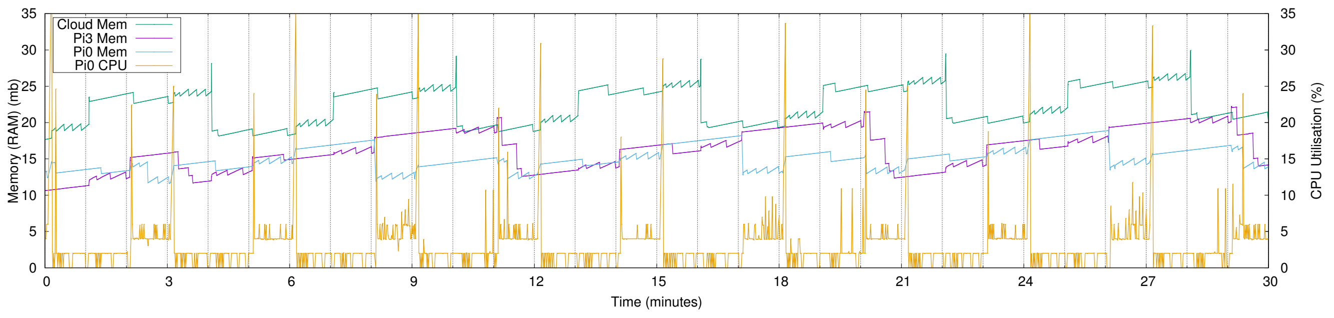


FIGURE 14 Multi-Hop Migration Analysis (*regulator* application)

the `navier-stokes` benchmark is that LAZYSNAP takes slightly longer than XPLICTGC because the snapshot step for LAZYSNAP takes more time. In XPLICTGC, the scope tree is explicit and thus always readily available for capture. If there are not many unused scopes in the program, the explicit GC procedure takes minimal time and the scope tree can be captured immediately. In LAZYSNAP, the scope tree is captured lazily, so a Mark-and-Sweep procedure is invoked to traverse and collect the active states, only upon a snapshot command. The results for `splay` bring out the opposite effect, in which XPLICTGC has to perform a lot of work to dereference the unused scopes before producing a snapshot. In LAZYSNAP, the unused scopes are left for the native GC, and the Mark-and-Sweep procedure simply captures the active scopes. The migration latency of LAZYSNAP on even the most constrained device (Raspberry Pi 0) is still quite reasonable (77ms for `splay`).

6.5 | Experiment 4: Multiple Migrations

In this experiment, we analyze the global behavior and performance of ThingsMigrate over time, when multiple migrations are performed between the *edge* and the *cloud*. More precisely, we analyze the effects of migrating a long-running asynchronous *service* that is not computationally expensive from one device to another. None of the benchmarks used in Experiment 2 fit this description, nor could we find publicly available JavaScript-based IoT benchmarks that satisfy this criteria (Section 6.1). Therefore, we developed and used our own benchmark – the *regulator* application that satisfies this criteria (by design). We first deploy the regulator application on our *cloud* server, then we migrate it to the *edge* devices (i.e., the Raspberry Pi 3 device, after one minute, and then to the Pi 0 device, after one minute). The application is then pushed back to the *cloud* server. This cycle is repeated 10 times (30 migrations over 30 minutes), and the CPU and memory utilization are measured in each instance.

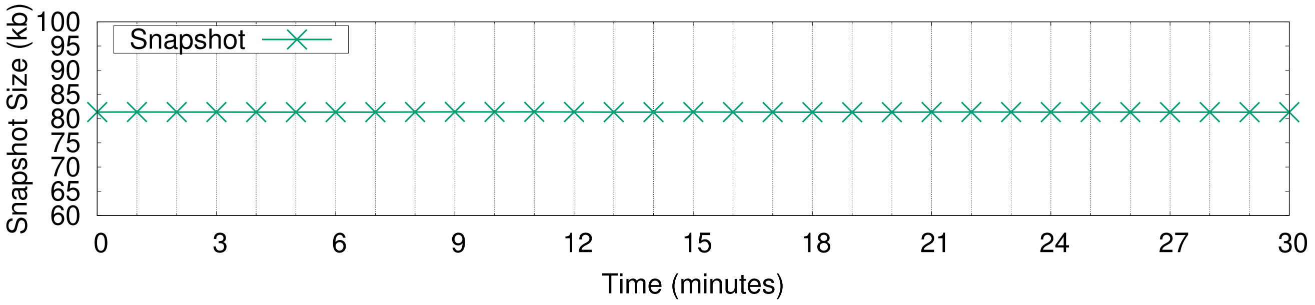


FIGURE 15 Snapshot Size over Time (*regulator* application)

The memory utilization results are shown in Figure 14, for the duration of the experiment (30 min). The migration cycles are denoted by a vertical bar (every 1 minute), and an oscillating variation pattern can be observed during the time periods for which each device was executing the regulator application. As can be observed, the memory usage fluctuates, for all devices, but remains overall stable, as each successive code restoration does not consume additional memory (assuming the memory needs of the application do not increase). The step-like appearance of the memory curves are explained by the JavaScript garbage collector (GC), which regularly claims small amounts of memory (i.e., during execution of the *regulator* – small pikes), and which periodically runs a more through collection (bigger drops). However, we also observe that the memory tends to very slowly increase over time, but this is not due to the multiple migrations – rather, this is an artifact of the experimental data collection process, which logs memory and CPU usage at a frequent interval (every 200ms) and keeps the data in memory. This is supported by Figure 15, which plots the snapshot size at each successive migration, which remains constant at 83kb. Finally, as in Experiment 2 (Section 6.3), the memory usage on the *cloud* server is higher than on the Pi devices.

The CPU usage is shown on the same Figure (14). For simplicity, we show CPU usage results only for one device (i.e., Pi 0, which is the most resource constrained), but the trend is similar on the others. As can be observed, the CPU usage peaks at about 4%-5% when the Pi 0 device is executing the application, and is close to 0% otherwise. The CPU usage during run-time remains constant across the different executions. The short spike *before* migration corresponds to the code reconstruction, and the short spike *after* migration corresponds to the serialization process, for which a small memory surge can also be observed.

6.6 | Summary

Overall, our results demonstrate that ThingsMigrate can enable the cross-platform migration of IoT JavaScript-based applications with acceptable performance overhead and without any modifications to the underlying VM. The run-time latency overhead using LAZYSNAP was 1.6% in the best case, and around 25% for control-yielding benchmark (*factorial*). Even for the compute-intensive, thread-blocking benchmarks from the Octane Suite, which are not representative of typical IoT workloads, ThingsMigrate was able to migrate them despite a higher run-time overhead. The memory overheads were more significant, ranging from 19% to 252%, though we believe that this is an acceptable tradeoff, given our approach to provide migration support purely at the application-level through code instrumentation.

Starting from the state-replication approach in TREECOPY, we explored the effects of different optimization techniques on the performance of the user application. XPLICTGC faired poorly in term of run-time overhead, despite a marginal reduction in memory consumption. This was mostly due to the explicit dereferencing of variables that have gone out-of-scope. In LAZYSNAP, we cut down memory consumption to nearly 1/3 of TREECOPY, by not keeping a replica of the internal state. We also saw a significant reduction in run-time overhead due to capturing the state lazily. Finally, our results show that ThingsMigrate can support multiple-hops of subsequent migration without altering the application semantics, while keeping the CPU and memory usage constant throughout and not adding any overhead over the migration cycle.

7 | CASE STUDY: MOTION DETECTOR

In this section, we describe our experience with using ThingsMigrate to build a realistic IoT application for video surveillance by adapting third-party JavaScript components developed for standalone Node.js applications. There is a strong motivation for

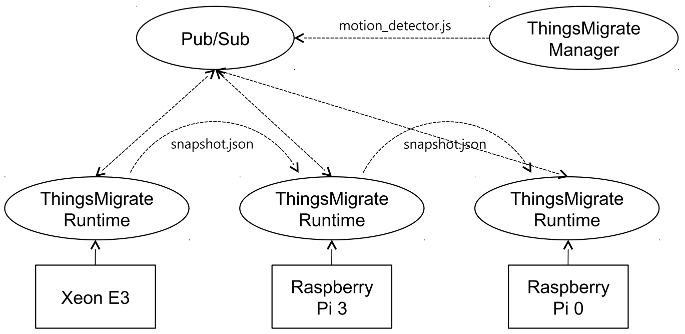


FIGURE 16 Case study setup

processing video streams at the *edge*[58,59], as sending live streams to the *cloud* for processing in real-time might not always be efficient or possible. The components that we adapted were not designed with ThingsMigrate in mind, and as a result, we had to make (minor) modifications to make them work with our system. We also evaluate this application using application-specific metrics, rather than CPU/memory usage (unlike Section 6).

7.1 | Experimental Setup

We set up an IoT network with four devices to build a surveillance system. Figure 16 shows the setup. We have used the TREECOPY technique for this case study, as the focus of the experiment was not on optimal performance but to understand how ThingsMigrate can integrate with a real-world application. The workflow for integrating XPLICTGC and LAZYSNAP remain the same.

The application logic is modularized into two components: a video streamer component that captures images from a video source such as a webcam, and a motion detector component that processes the images to detect motion. Unlike the video streamer, which is bound to a single device by the peripheral from which it needs to capture video, the motion detector can be run on any device as it performs computations on the image data. We measured the behavior of the system over a series of migrations of the motion detector across the three systems (Raspberry Pi 3, Pi 0, and the *cloud* server from Section 6).

Video-streamer: We used `FFmpeg`[60], a popular open-source software for handling multimedia, to capture individual frames from a video stream. For the purpose of the experiment, the component was configured to stream from a video file instead of a peripheral such as a webcam, so that we have a deterministic and reproducible sequence of frames. To interface with the `FFmpeg` process from the JavaScript layer, we adapted a third-party NPM library called `fluent-ffmpeg`[61], which we use to capture individual frames. We then publish them over the Pub/Sub interface. The capture-and-publish routine was written as a single JavaScript function `captureFrame` that was passed into a `setInterval` call with interval set to 200ms (i.e., a rate of 5 frames per second). We used the *cloud* server to serve as a surveillance camera and run the video streamer component.

Motion Detector: This component was written entirely in JavaScript without having to interface with any external software. We integrated a third-party NPM module called `jimp`[62], which provides an API to read `Buffer` objects (i.e., received from the Pub/Sub overlay) and perform image processing tasks. The component stores binary frame data for the *n* latest frames.

The motion detection logic (i.e., function `detectMotion`) iterates through the array of images and computes the difference between subsequent frames by calling `jimp.diff()`. The binary difference between the frames is published over the Pub/Sub interface. In addition, if more than 10% of the pixels are altered, an alert message is also published under a different topic. The `detectMotion` function is passed to a `setInterval` call with the interval set to 500ms - this is lower than the frame rate of the video streamer (Section 7.2 explains why). Since the `detectMotion` works by retrospective inspection of past frames, the array of `Buffer` objects containing the image data needs to be migrated. Otherwise, the restored component would need to wait for the buffer of past frames to be filled again – thereby skipping the motion detection process for a given time window, and missing potentially important motion.

Although we do not fully support the migration of external libraries (Section 5.1.4), it was possible to integrate the third-party NPM libraries as the objects they created were native JavaScript objects and the API calls were limited to stateless operations. For instance, since the Buffer objects are native objects, they could be easily serialized and migrated. The function call to

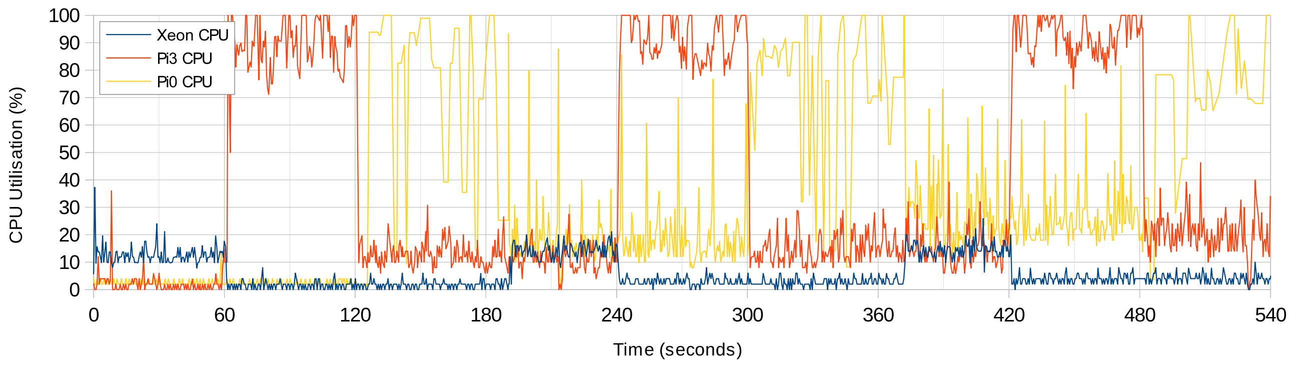


FIGURE 17 CPU Usage over time – Motion Detector component

`jimp.diff()` is a stateless operation, since it does not create any additional scopes and its execution context is destroyed after it returns. Such stateless operations do not affect the migration process because we do not need to serialize their scopes.

Finally, we collected performance statistics by subscribing to a Pub/Sub topic at which each ThingsMigrate runtime publishes its CPU and memory usage. To monitor and verify that the motion detection was working correctly, we used the ThingsJS web dashboard, which displays the images by converting the data into a base64 encoded PNG image.

7.2 | Results

To automate our migration test in a controlled fashion, we wrote a Node.js script, using the *ThingsMigrate Migrator* to send commands to the IoT devices over the Pub/Sub interface. We sent a `migrate` command every 1 minute to the *cloud* Server, Pi 3, and Pi 0, in that order, and back. We repeated the cycle 3 times.

Figure 17 shows the CPU usage over time as the application is migrated between the devices. The collected data for CPU and memory usage across the three devices exhibit a similar pattern to the regulator component discussed in Section 6.4. The CPU usage on a device has a spike upon receiving a snapshot and just before sending a snapshot, remains high while it is running a component, and stays near 0 during the idle state. The memory consumption stays within a narrow range, with the garbage collector being triggered more frequently while a device is running a component, and only occasionally while it is idle.

However, on the Raspberry Pi 0's console, we observed error messages showing that the process failed at regular intervals. This is because the asynchronous call to `detectMotion` took much longer than the set interval of 500ms, due to the limited computational capacity of the Pi0, which led to the event queue filling up faster than the JavaScript VM could consume, which eventually led to overflowing and halting of the program.

Figure 18 shows the frame rate measured in Frames Per Second (FPS) on each device over time. The FPS was calculated using the formula $\frac{1}{\Delta t}$ where Δt is the time taken to execute the `detectMotion` function. The figure shows the FPS dropping below the required FPS of 2 over the periods between 120 and 180, 300 and 360, and 480 and 540 seconds, during which the Pi 0 was running the motion detector component. We can also observe the `detectMotion` function blocking the thread at 180 seconds and 360 seconds, preventing the migration from being triggered. The FPS drops below 2 occasionally during Pi 3's execution, but for most frames it is able to process within the time interval, compensating for the delay overall.

In summary, we were able to successfully migrate a third-party application with minimal modifications between different devices. We also found the frame rate measured in FPS was acceptable in most cases for the application.

8 | DISCUSSION

Equivalency of 3 versions. Since we instrument the original code and transform the program, we technically change the semantics of the program. From an operational semantics perspective, the instrumented program is not strictly equivalent (e.g.,

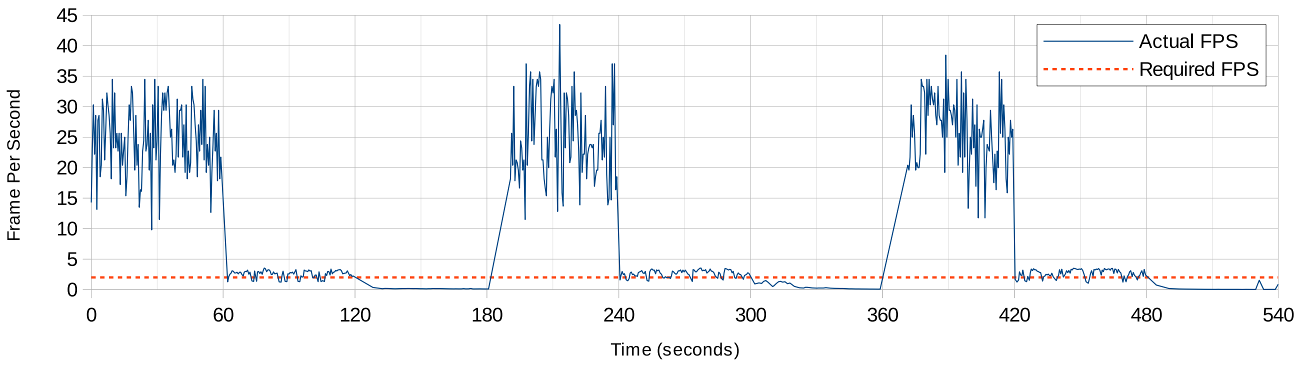


FIGURE 18 FPS over time – Motion Detector component

alpha-equivalent) to the original. That said, we use the relaxed notion of *observational equivalence*, where we claim that 2 programs are equivalent if their observable outputs are indistinguishable. This is a practical and acceptable notion of equivalence in the context of real-world usage, as evidenced by the use of regression tests over a software life-cycle[63,64,65].

Based on this reasoning, we verify the safety of our instrumentation by observing the behaviour of the instrumented program and comparing it against the behaviour of the original program using a set of unit tests and micro-benchmarks. For example, to verify that we do not break the closure semantics, we test a minimal program that defines a single closure `function foo()` containing a local variable `x` initialized with an arbitrary value, then try to print the value of `x` from the global scope – a correctly transformed program should print `undefined`. We use a set of such minimal programs and more complex programs (e.g., nested closures, multiple instances of closures) to test each of the JavaScript semantics, ranging from simple variable assignment and loops to prototypal inheritance. These tests also help us assess the completeness of our instrumentation, as failed tests reveal the JavaScript features we do not support. Using this rubric, LAZYSNAP is the most complete approach, followed by TREECOPY, which does not fully support migrating prototypes. As we have mentioned, XPLICTGC actually breaks the closure semantics when dealing with multiple nested closures. However, we have not pursued to correct XPLICTGC as we have abandoned the approach after discovering its relatively poor performance.

Security Impact. We examine the security impact of transforming a program, since the instrumented program exposes to ThingsMigrate the closed variables, which otherwise would be hidden to a third-party script running in the same VM. Firstly, we assume that the user (or a group of trusted users such as a company) has control over the host platform on which ThingsMigrate Runtimes run, and that the process snapshot is transfered over the Internet. This assumption applies to most Web services, as the service providers either own the host machines or rent VMs from trusted parties, and data is transferred between servers over the Internet. Given this assumption, we claim that the attack surface of ThingsMigrate is fundamentally no larger than an ordinary Web service; it is secure as long as we apply the standard Web security practices of encrypting communication channels and protecting the hosts[66,67,68].

There are 2 attack vectors through which a closed variable can be leaked: 1) snapshot image, and 2) a third-party script. The first attack vector is straightforward – the snapshot contains the values of all the variables and it is transferred over the network. Thus, to preserve the confidentiality of the snapshot, we must ensure that the communication is secure via end-to-end encryption between the ThingsMigrate Runtimes. While we have not secured the communication channels in our implementation, it is trivial to do so – because we control both communication endpoints – by leveraging existing solutions such as TLS. As per the second attack vector (via third-party script), we first summarize ThingsMigrate's operation: ThingsMigrate itself is written as a JavaScript program that runs inside a VM like Node.js, exposes a user interface at a public-facing network port through which a user sends code to execute, and instruments and executes a given program upon user request. At the user interface, we must ensure that the user is trusted and the communication channel between the user and ThingsMigrate is secure. While we do not address this aspect in our implementation, it can easily be addressed by using existing Web security solutions such as standard authentication, OAuth, or PKI (SSL certificates). This provides the bare minimal protection from a technical standpoint, but the system is still vulnerable from authenticated users with malicious intent – i.e., compromised clients. For example, an honest user's program could be vulnerable to a program sent by a malicious user, similarly to a cross-site scripting attack. TREECOPY was vulnerable to this attack, as the closed variables are directly reachable from a global object and the user programs were run in the same process space. In LAZYSNAP, this attack is more difficult, firstly because we do not expose closed variables in the global scope, and secondly because each program is run as a child process of the ThingsMigrate Runtime, and thus is isolated from each other. The only channel a malicious program has to another program is through the ThingsMigrate API – by invoking the `snapshot` function and receiving the snapshot. The current implementation of ThingsMigrate is vulnerable to this particular attack, but we can defend against it by implementing permission control. For example, each program would have an *owner*, and migration control would be permitted only by the *owner*. Albeit the lack of implementation, the defense we have described – such as using TLS, authentication, and permission control – are standard practice in Web-based systems, and hence we claim that ThingsMigrate is on par with ordinary Web services in terms of security.

Algorithmic Complexity. In terms of algorithmic complexity, we first point out that the *input instance* is the user program and its size is simply the size of the JavaScript code in UTF8. The algorithm can be broken down into 3 phases: parsing phase, AST modification phase, and code generation phase. The algorithmic complexity of the parsing phase is $O(n)$, as JavaScript's lexical grammar is Left-to-Right (LR)[45] and the parser is essentially a single-pass stream operator over a text stream. We use the `esprima` parser[55] whose algorithmic complexity is undocumented, but we assume $O(n)$ which is typical of LR parsers[69,70]. After parsing, ThingsMigrate modifies the relevant AST nodes by recursively processing the tree starting from the root. TREECOPY and XPLICTGC were not context-free, as they had to track each variable name by looking up the surrounding nodes to determine the name's lexical scope. In the worst case, the algorithmic complexity would be $O(n^2)$, as the whole tree could be looked up for each node. However, LAZYSNAP modifies the AST in a nearly context-free manner; it builds the local context of each function expression as it traverses its body, and then injects a new node as it exits the node. Thus, the modification phase also completes in $O(n)$. Finally, the code generation phase is the reverse process of parsing, and it also completes in $O(n)$; we use the `escodegen` library[56]. Overall, the algorithmic complexity of ThingsMigrate's code instrumentation is therefore $O(n)$. We also observed this experimentally in Figure 9a.

9 | CONCLUSION AND FUTURE WORK

In this paper, we presented ThingsMigrate, an approach that enables platform-independent migration of stateful JavaScript processes across diverse IoT devices. ThingsMigrate transparently instruments the application code before executing it, injecting application-level objects that expose the hidden states of a JavaScript program such as local variables captured inside closures. Additional event listeners are injected into the instrumented program to provide an interface for serializing the program state into a snapshot during run-time. Given a platform-agnostic representation (i.e., JSON snapshot) of the process state, ThingsMigrate generates code that restores the state of the program and resumes execution. Further, ThingsMigrate enables multiple subsequent migrations by formulating the snapshot procedure and restoration procedure as inverses of each other, to ensure that the program before capture and after restoration are semantically equivalent. ThingsMigrate is purely an application-level technique relying only on the semantics of the language, and requires neither any modification to the underlying VM nor the platform.

We presented 3 different versions of ThingsMigrate, each with different tradeoffs. In the first TREECOPY approach, we maintain an explicit replica of the program state by copying any state variables into the injected `Scope` objects. In an attempt to reduce the memory footprint, we carry out a more invasive program transformation in XPLICTGC, in which we convert all lexically-scoped variables into explicit properties of `Scope` objects. Consequently, we face additional difficulties having to explicitly dereference out-of-scope variables, which eventually degrades performance during run-time. Combining the learning outcomes from TREECOPY and XPLICTGC, we devise an optimized technique in LAZYSNAP, where we avoid both keeping an explicit replica of the state and managing the `Scope` life-cycle. LAZYSNAP introduces minimal run-time overhead by capturing the state *lazily via callbacks*, and by aligning its `Scope` hierarchy with the original program's scope hierarchy, thus delegating to the native GC the management of the `Scope` life-cycle.

We evaluated each version of ThingsMigrate on three different devices (IoT and *cloud*), using both Chrome Octane benchmarks and custom benchmarks. The results show that ThingsMigrate can instrument user programs within microseconds on a *cloud* device, and migrate processes (i.e., capture snapshot and generate restored program) between devices within reasonable time bounds. ThingsMigrate imposes an average of 33% execution time overhead during run-time for non-blocking IoT applications, which is reasonable, and does not lead to significant slowdown. Finally, we show that ThingsMigrate supports multiple subsequent migrations across heterogeneous devices without incurring additional overheads.

As future work, we intend on improving support for more complex cases of classes and prototypes, as well as supporting the features of the newer ECMA standards out-of-the-box (i.e., without the use of a transpiler). We would also like to accomplish migration without interrupting the execution flow (i.e., seamless migration). Other potential extensions could be to adapt our approach to provide fault tolerance in an IoT setting, or providing application *replication* onto several devices, for increased performance and dependability.

ACKNOWLEDGEMENTS

This work is supported by a research gift from Intel, a Discovery grant and Post-Doctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Institute for Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia (UBC). We thank the anonymous reviewers of ECOOP 2018 and the SPE journal for their helpful comments that have improved this paper.

References

1. Saha HN, Mandal A, Sinha A. Recent trends in the Internet of Things. In: IEEE. ; 2017: 1–4.
2. Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 2013; 29(7): 1645–1660.
3. Taivalsaari A, Mikkonen T. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 2017; 34(1): 72-80. doi: 10.1109/MS.2017.26
4. 2017 TIOBE Index. https://www.tiobe.com/tiobe-index/; 2017.
5. Gavrin E, Lee SJ, Ayrapetyan R, Shitov A. Ultra Lightweight JavaScript Engine for Internet of Things. In: ACM; 2015; New York, NY, USA: 19–20.
6. Intel XDK. https://software.intel.com/en-us/xdk; 2017.
7. Guinard D, Trifa V. Towards the web of things: Web mashups for embedded devices. In: . 15. ; 2009.
8. Gascon-Samson J, Rafiuzzaman M, Pattabiraman K. ThingsJS: Towards a Flexible and Self-adaptable Middleware for Dynamic and Heterogeneous IoT Environments. In: M4IoT '17. ; 2017: 11–16.
9. Chaniotis IK, Kyriakou KID, Tselikas ND. Is Node. js a viable option for building modern web applications? A performance evaluation study. *Computing* 2015; 97(10): 1023–1044.
10. Lin J, El Gebaly K. The Future of Big Data Is... JavaScript?. *IEEE Internet Computing* 2016; 20(5): 82–88.
11. Tilkov S, Vinoski S. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 2010; 14(6): 80–83.
12. Vaarala S. DukTape. http://www.duktape.org/; 2017.
13. mjs. https://github.com/cesanta/mjs; 2017.
14. Sin D, Shin D. Performance and Resource Analysis on the JavaScript Runtime for IoT Devices. In: Springer. ; 2016: 602–609.
15. Node-RED Website. https://nodered.org/; 2017.
16. Services AW. IoT Greengrass. https://docs.aws.amazon.com/greengrass/; 2019.
17. Satyanarayanan M. The Emergence of Edge Computing. *Computer* 2017; 50(1): 30-39. doi: 10.1109/MC.2017.9
18. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 2016; 3(5): 637–646.
19. Levy A, Campbell B, Ghena B, et al. Multiprogramming a 64kB Computer Safely and Efficiently. In: ACM. ; 2017: 234–251.

20. Gascon-Samson J, Jung K, Goyal S, Rezaiean-Asel A, Pattabiraman K. ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications. In: Millstein T., ed. *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2018; Dagstuhl, Germany: 18:1–18:33

21. Lo JTK, Wohlstadter E, Mesbah A. Imagen: Runtime Migration of Browser Sessions for Javascript Web Applications. In: WWW '13. ACM; 2013; New York, NY, USA: 815–826.

22. Kwon Jw, Moon SM. Web Application Migration with Closure Reconstruction. In: WWW '17. ; 2017; Geneva, Switzerland: 133–142.

23. Wright A, Andrews H. Json schema: A media type for describing json documents. In: 2017.

24. Biswas S, Sharif K, Li F, Latif Z, Kanhere SS, Mohanty SP. Interoperability and Synchronization Management of Blockchain-Based Decentralized e-Health Systems. *IEEE Transactions on Engineering Management* 2020: 1-14.

25. Bellucci F, Ghiani G, Paternò F, Santoro C. Engineering JavaScript state persistence of web applications migrating across multiple devices. In: ACM. ; 2011: 105–110.

26. Oh J, Kwon Jw, Park H, Moon SM. Migration of Web Applications with Seamless Execution. In: VEE '15. ACM; 2015; New York, NY, USA: 173–185

27. Cruysse V. dJ, Hoste L, Van Raemdonck W. FlashFreeze: Low-overhead JavaScript Instrumentation for Function Serialization. In: META 2019. ACM; 2019; New York, NY, USA: 31–39

28. Foundation M. Rhino - Mozilla. https://github.com/mozilla/rhino; 2020.

29. Andrica S, Candea G. WaRR: A tool for high-fidelity web application record and replay. In: IEEE. ; 2011: 403–410.

30. Mickens JW, Elson J, Howell J. Mugshot: Deterministic Capture and Replay for JavaScript Applications.. In: . 10. ; 2010: 159–174.

31. Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: ACM. ; 2013: 488–498.

32. Burg B, Bailey R, Ko AJ, Ernst MD. Interactive record/replay for web application debugging. In: ACM. ; 2013: 473–484.

33. Alimadadi S, Sequeira S, Mesbah A, Pattabiraman K. Understanding JavaScript event-based interactions. In: ACM. ; 2014: 367–377.

34. Milojičić DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S. Process Migration. *ACM Comput. Surv.* 2000; 32(3): 241–299. doi: 10.1145/367701.367728

35. Zarrabi A. A generic process migration algorithm. *International Journal of Distributed and Parallel Systems* 2012; 3(5): 29.

36. Zhongyuan S, Jianzhong Q, Shukuan L, Qiang Z. Use Pre-record Algorithm to Improve Process Migration Efficiency. In: ; 2015: 50-53

37. Wang C, Mueller F, Engelmann C, Scott SL. Proactive process-level live migration in HPC environments. In: IEEE Press. ; 2008: 43.

38. Litzkow M, Tannenbaum T, Basney J, Livny M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. tech. rep., University of Wisconsin-Madison Department of Computer Sciences; 1997.

39. Wang S, Urgaonkar R, Zafer M, He T, Chan K, Leung KK. Dynamic service migration in mobile edge-clouds. In: ; 2015: 1-9.

40. Machen A, Wang S, Leung KK, Ko BJ, Salonidis T. Live Service Migration in Mobile Edge Clouds. *IEEE Wireless Communications* 2018; 25(1): 140-147.

41. Ha K, Abe Y, Chen Z, et al. Adaptive VM handoff across cloudlets. *Technical Report CMU-CS-15-113* 2015.

42. Clark C, Fraser K, Hand S, et al. Live migration of virtual machines. In: USENIX Association. ; 2005: 273–286.

43. Wood T, Shenoy PJ, Venkataramani A, Yousif MS, others . Black-box and Gray-box Strategies for Virtual Machine Migration.. In: . 7. ; 2007: 17–17.

44. International E. *ECMAScript Language Specification*. Ecma International. 5.1 ed. 2011.

45. International E. *ECMAScript 2015 Language Specification*. Ecma International. 6.0 ed. 2015.

46. Babel.js JavaScript Compiler. https://babeljs.io/; 2017.

47. Karagiannis V, Chatzimisios P, Vazquez-Gallego F, Alonso-Zarate J. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing* 2015; 3(1): 11–17.

48. Eugster PT, Felber PA, Guerraoui R, Kermarrec AM. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 2003; 35(2): 114–131. doi: 10.1145/857076.857078

49. Yoon Y, Muthusamy V, Jacobsen HA. Foundations for highly available content-based publish/subscribe overlays. In: IEEE. ; 2011: 800–811.

50. Chang T, Meling H. Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure. In: IEEE. ; 2012: 454–456.

51. Crockford D. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc." . 2008.

52. Andreasen E, Gong L, Møller A, et al. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys (CSUR)* 2017; 50(5): 66.

53. Madsen M, Lhoták O, Tip F. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages* 2017; 1(OOPSLA): 86.

54. Wilson PR. Uniprocessor garbage collection techniques. In: Springer. ; 1992: 1–42.

55. Hidayat A. esprima: ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/; 2017.

56. Suzuki Y. escodegen: ECMAScript code generator. https://github.com/estools/escodegen; 2017.

57. Chromium Octane Benchmark Suite. https://github.com/chromium/octane; 2017.

58. Wang J, Feng Z, Chen Z, et al. Bandwidth-efficient live video analytics for drones via edge computing. In: IEEE. ; 2018: 159–173.

59. Ananthanarayanan G, Bahl P, Bodík P, et al. Real-time video analytics: The killer app for edge computing. *computer* 2017; 50(10): 58–67.

60. FFmpeg Website. https://www.ffmpeg.org; 2017.

61. Fluent ffmpeg-API for node.js. https://github.com/fluent-ffmpeg/node-fluent-ffmpeg; 2017.

62. Jimp: JavaScript Image Manipulation Program. https://github.com/oliver-moran/jimp; 2017.

63. Harrold MJ, Jones JA, Li T, et al. Regression Test Selection for Java Software. In: OOPSLA '01. Association for Computing Machinery; 2001; New York, NY, USA: 312–326

64. Csallner C, Smaragdakis Y. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 2004; 34(11): 1025–1050.

65. Wong WE, Horgan JR, London S, Agrawal H. A study of effective regression testing in practice. In: ; 1997: 264-274.

66. Garfinkel S, Spafford G. *Web security, privacy & commerce*. " O'Reilly Media, Inc." . 2002.

67. Allen J. Cert system and network security practices. In: ; 2001: 22–24.

68. Yue C, Wang H. Characterizing Insecure Javascript Practices on the Web. In: WWW ’09. Association for Computing Machinery; 2009; New York, NY, USA: 961–970

69. Knuth DE. On the translation of languages from left to right. *Information and control* 1965; 8(6): 607–639.

70. Aho AV, Sethi R, Ullman JD. Compilers, principles, techniques. *Addison wesley* 1986; 7(8): 9.

How to cite this article: K. Jung, J. Gascon-Samson, S. Goyal, A. Rezaiean-Asel, and K. Pattabiraman (2020), ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications, *Q.J.R. Meteorol. Soc., 2017;00:1–6*.